Analysis and Verification of Service Contracts

Okika, Joseph

Publication date:
2010

Document Version
Accepteret manuscript, peer-review version

Analysis and Verification of Service Contracts

Joseph C. Okika

Aalborg University
Department of Computer Science

Analysis and Verification of Service Contracts

Joseph C. Okika

Ph.D. Dissertation

March, 2010

Aalborg University
Department of Computer Science

ABSTRACT

Service contracts are specifications used by interacting services to agree on ways of interaction to achieve an intended business goal. This raises a number of interesting questions. What languages are used to express service contracts? What are the semantics of the languages? Do the connected services adhere to the contract and behave as expected? Are the specifications consistent? To answer these and other questions, this work reports on a study of service contract specification languages, define semantics of a particular language, and analyze behavioural specification of service contracts using simulation and model checking techniques.

Chronologically, the first contribution is an automated technique to check consistency properties across specifications. The results of the consistency checks reveal that the derived processes from the two service contract specifications (WS-CDL and BPEL) are bisimilar and trace equivalent only when certain intricate features, e.g., fault handling, is hidden. Therefore we note that WS-CDL can only be consistent with an abstract version of BPEL without the presence of some intricate features. In the second contribution, we derive semantic models in the form of timed automata and check the absence of deadlock for WS-CDL and BPEL service contract specifications. We check the property that the system is able to progress from start to termination and for a BPEL process, we check that the methods can be executed satisfying the contracts or generating the exceptions. Other contributions include: a classification of languages and standards based on the aspects of service contracts they cover; a structural operational semantics of BPEL activities; a prototype executable semantics of BPEL based on Rewriting Logic and Maude; and a translation process that maps a BPEL process to UppAal timed automata for automatic verification.

In conclusion, we argue that introducing (automated) analyses and verification of service contracts at the design and development phase before testing and deployment will improve the reliability of service-based systems. The analyses prototyped here improve our understanding of the behaviour of connected services.

DANSK SAMMENFATNING

Servicekontrakter er specifikationer, som bruges mellem programmer i en serviceorienteret arkitektur til at beskrive, hvordan de skal samvirke for at opnå et bestemt forretnings mål. Dette rejser en række interessante spørgsmål: Hvilke sprog anvendes til at udtrykke servicekontrakter? Hvad er semantikken af disse sprog? Opfører de tilknyttede tjenester sig som forventet i forhold til kontrakten? Er specifikationerne konsistente? For at besvare disse og andre spørgsmål, rapporterer dette arbejde om en undersøgelse af specifikationssprog for servicekontrakter, definerer semantik af et bestemt sprog, og analyserer specifikationer af servicekontrakter med simulerings- og modelcheck-teknikker.

Det første bidrag er en automatiseret teknik for at kontrollere konsistens mellem specifikationer. Resultaterne af konsistenskontrollen viser at afledte processer fra to servicekontrakt specifikationer (WS-CDL og BPEL) kun bisimilære og sporækvivalente når visse komplekse opførsler, f. eks. fejl håndtering, er skjulte. Vi kan derfor konstatere, at WS-CDL kun er overensstemmelse med en abstrakt version af BPEL uden tilstedeværelsen af visse opførsler. I et andet bidrag udleder vi semantiske modeller i form af tidsautomater og kontrollerer for baglåse i WS-CDL og BPEL servicekontrakter. Vi kontrollerer også at systemet er i stand til at bevæge sig fra start til terminering og for at BPEL metoder opfylder kontrakter eller generer en fejlmelding. Andre bidrag er en klassificering af sprog og standarder baseret på de aspekter af servicekontrakter som de dækker; en strukturel operationel semantik af BPEL aktiviteter; en prototype udførbar semantik af BPEL baseret på Rewriting Logic og Maude; en oversættelsesproces fra BPEL til UppAal tidsautomater for automatisk kontrol.

Vi konkluderer at indførelse af (automatiserede) analyser og kontrol af servicekontrakter i design og udviklingsfasen inden afprøvning og ibrugtagning vil forbedre pålidelighden af servicebaserede systemer. De analyserede prototyper vil forbedre forståelsen for opførslen af samvirkende tjenester.

To my family and friends for all their support

Acknowledgements

My sincere appreciation goes to Prof. Anders P. Ravn. He deserves my gratitude for his insight, time, and effort in guiding this thesis in addition to his support over the years in matters not only in work, but life in general.

I would like to express my gratitude to my co-authors especially Emilia, Gordon, Stephen, Gerardo, Olaf, and Cristian. Thanks to Zhiming Liu, Volker Stolz and the members of rCOS group at the United Nations University - International Institute for Software Technology, Macau.

Thank you Prof. Olaf Owe for your time, contribution and hospitality during my stay at Oslo as well as for introducing me to Rewriting Logic and Maude. My special thanks to Cristian and Ioana for their kindness. Thanks go to Gerardo, Martin, Einar and the entire PMA group for their friendship and support during my stay at UiO.

My special thanks to Emilia, Gregorio, Valentin and the members of Real Time and Concurrent Systems group for their hospitality during my visit to the University of Castilla-La Mancha. Thanks are also extended to Liang-Jie Zhang and Nianjun Zhou (Joe) for hosting me at the IBM Watson Research Center, Hawthorne. Thanks to Larry Mikulski of ActiveVOS for the success stories and discussion of service orchestration and BPEL.

Thanks to the secretaries at the department, especially Helle S., Helle W., Ulla, Lene, Ulla L. and Joan, for all their help with practical and administrative issues during these years.

Thanks to my colleagues Istvan, Henrik, Saleem, Shuhao, Marius, and Mehdi. I also thank the Professors, staff and members of DES, CISS and Computer Science Department. Thanks to Rico and Kim for taking me to football trainings and matches. I would like to thank my brother Ken and numerous friends who helped me throughout this academic exploration.

The financial support of the "Contract-Oriented Software Development for Internet Services" (COSoDIS) project is gratefully acknowledged.

I am grateful to you all!

Joseph C. Okika
March 2010

Contents

Chapter 1

Introduction

Software applications based on Internet services and Service Oriented Architecture (SOA) [Erl05] are growing in complexity. SOA offers a tremendous opportunity to connect services to achieve a particular business goal, because services are inherent in many business applications such as Paypal, E-bay, Google services as well as social network applications such as Facebook, Myspace, etc. As an architectural style, SOA has a profound impact on the overall service implementation, composition and integration, because Web Services are loosely coupled and therefore can be reused easily. This architectural approach is particularly applicable when multiple applications running on varied technologies and platforms need to communicate with each other. In this way, enterprises can mix and match services to perform business transactions with less programming effort.

In Service Oriented Architecture, services are first class citizens and are autonomous as well as distributed in nature. They can be connected to form higher level services or applications to solve business goals. Scientifically, these connected services could be seen as communicating sequential processes, exhibiting certain patterns of concurrent interaction. Of course, this concurrent interaction raises a lot of issues and problems such as managing connected services, monitoring their interaction, analyzing the behavior of interacting services, verifying the functionality of individual services as well as connected services. In summary, connected services are complex because they combine communication and concurrency and this is the main motivation for developing analysis techniques based on recent development in tools for formal analysis such as model checkers and verification assistants.

Examples

An example of a service-based critical system is found in *The European Organization for Nuclear Research (French: Organisation Européenne pour la Recherche Nucléaire)*, known as CERN. The world's largest particle physics laboratory has embraced SOA and BPEL orchestration for automating/developing their safety and logistic processes. Here is a quote from CERNs Derek Mathieson [Law09].

> For example, if there is an engineer who wants to come and work in a particular area and the work that the engineer does is welding or something like this, it may cause smoke, and then the engineer will have to disable the fire detection systems. So there is a business process to make sure the fire detection system is switched off, the work is done, and then the fire alarm is switched back on again. This is what we would call a business process. We have to make sure that the appropriate people are informed, that there is follow-up to make sure that when the process reaches the expected end date that someone actually will check and make sure the fire alarms are switched back on again, and this kind of thing. ... So the idea of having a more service-oriented architecture, which allows us to have a standardized interface to each of the different systems and automate everything using the BPEL-based process engine, was a fairly nice match for us.

Here a BPEL orchestration is used to manage a fire detection system in the Nuclear research laboratory.

An example of a system with great monetary value is an Excise Movement (Monitoring) Control System (EMCS) [Viv09], which manages International (28 EU member states) Cross-Border tax and revenue. The EMCS specifications are intended to be used by member nations to implement systems that replace manual paper custom and excise processes. There are 50 or so processes that are specified by EMCS. Process orchestration is the heart of the system and it is based on SOA and BPEL.

Background for SOA

These examples show that BPEL based systems may be around for some years. The emergence of SOA can be traced back to the idea of building software from components: Component-based software engineering (CBSE), also known as component-based development (CBD). Furthermore, a software

component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties [Szy02]. This solution points to a paradigm shift where software systems are built from standard components rather than developed from scratch. This move towards building systems from standard components introduces some complex mechanisms as found in the major component standards such as COM, CORBA, and EJB [EK02]. Multi-threading is one of the key mechanisms. Multi-threading is the concept of supporting multiple sequential activities concurrently over the same state space. This increases the complexity of components, especially when dealing with true concurrency. For example, if component instances execute on separate processors, then concurrent requests need to be handled. Complete locking of a component instance while one request is handled is possible, but may lead to deadlocks or poor response time. Further, it is difficult and perhaps impossible in the general case to specify exactly how an extension and its synchronization and concurrency needs may interfere with those of the base system without causing unwanted effects [Szy02].

SOA like CBD, has communication and concurrency concerns coupled with remote access which additionally leads to an issue of trust. However, trust cannot be guaranteed because of the combination of communication, concurrent and remote access mechanisms found in SOA systems. Further, the service implementation is not known to the consumer, since only the interface is exposed for invocation. This leads to considering the notion of service contract where the interacting parties can specify an agreed way (protocol) of interaction to achieve a specified business goal. The potential value of service contracts is that a form of co-ordination can be enforced on how the composed (participating) services interact as well as measuring their performance.

A service operates under a contract/agreement which will set expectations, and a particular ontological standpoint that influences its semantics [PL03]. These expectations make it increasingly important that these services are analyzed at development time based on their contracts (for instance, expected behaviour) to improve the reliability of software applications built using such services. One obvious reason is because deploying/executing a contract violating system consumes a lot of resources.

Service contracts deal with aspects such as behaviour, functionality, quality of service and security [OR08]. Functionality deals with what functions the services provide. Behaviour defines the interaction protocol for collaborating services. Extra-functional properties, for instance Quality of Service (QoS), give constraints on the values acceptable for a concrete service. Secu-

rity is often specified by a protocol (a behaviour) combined with encryption functions. However, it is hard to create a "one size fits it all" contract where all the aspects are embodied, because different aspects are covered by different notations and technologies.

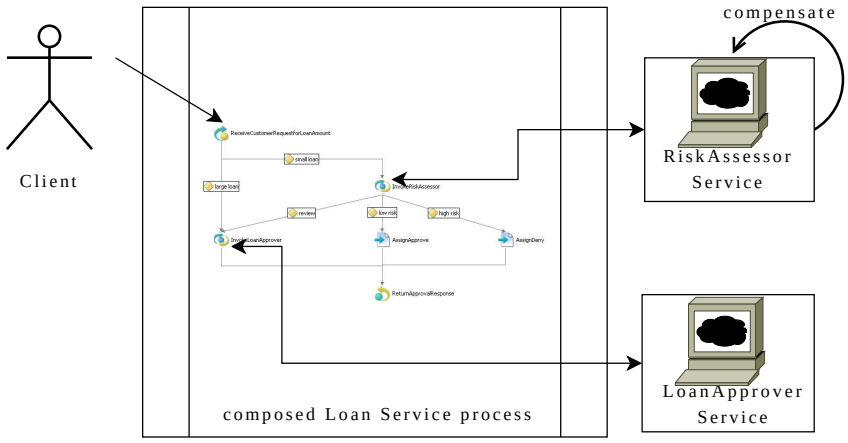


Figure 1.1: A composite loan approval service.

Consider the typical service-based system illustrated in Figure 1.1. It consists of two external services interacting to achieve a particular business goal. The composite service interacts with the Risk Assessment and Approval services to provide a one stop access to clients who wish to obtain loans. There are many interactions between the composite service and the external services, raising many concerns such as the behaviour of the composite service particularly in the presence of faults and exceptions. In fact, it has several concurrent interactions and hence the level of complexity is higher than a "normal" client-server application.

We can test the system (with several kinds of testing techniques) before deployment. However, testing the behaviour of the connected services becomes complicated because the order in which the operations of each service are called must be considered. Further, another difficulty arises because of the explosion in the number of possible paths taken by each service during execution. There could also be a possibility of deadlocks and race conditions when services are connected.

Consider a use case in the example shown in Figure 1.1, where there are three services: Loan process, RiskAssessor, and LoanApprover. When a client makes a request to the Loan process, it can invoke the operations of RiskAssessor. This may lead to invocation of the LoanApprover where a response from the RiskAssessor is yet to arrive. The LoanApprover will be expecting the requested information. Thus, the Loan process waits for a

response from RiskAssessor while LoanApprover waits for the Loan process. This causes the services to be deadlocked. A second example is the use of call back mechanisms. For instance, the Loan Approver invokes the Loan process and waits. The Loan process makes a call-back to the Loan Approver and waits for a response. In this situation, both are in a deadlock.

It is therefore desirable to complement testing with analysis techniques to analyze the behaviour of connected services. For example, one can analyze that there are neither deadlocks nor livelocks that may cause abnormal behaviour of the connected services no matter the different responses from each of the external services. In our work, we focus on the behaviour aspects of service contracts which define the interaction protocol of collaborating services, because they combine communication and concurrency and are features that are hard to test with standard testing technologies.

SOA Application Development Processes

Development of SOA applications just like other software development go through phases or life cycle. Several approaches and methodologies ([PH06, PvdH07]) exist that utilize similar or varied phases in developing SOA applications. A detailed survey of Service oriented methodologies can be found in [RDS07]. We consider a few that use similar phases in order to point where we want contribute in the development of SOA applications in order to improve reliability.

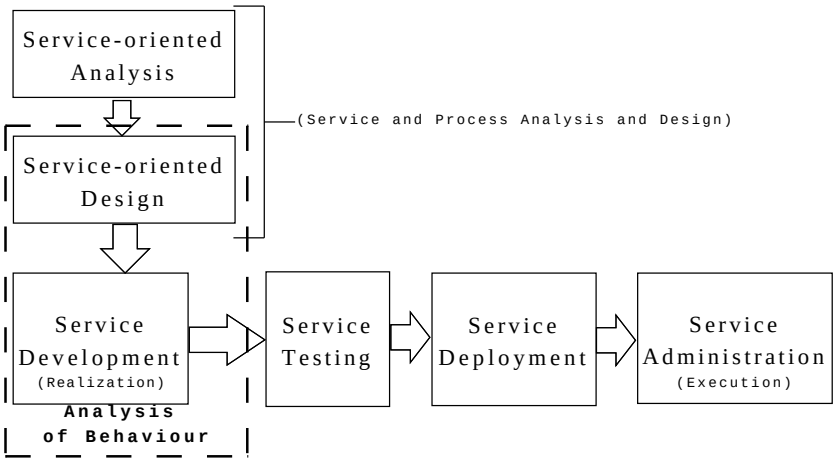


Figure 1.2: SOA application development methodology/phases.

Figure 1.2 shows the phases adopted in [Erl05, PvdH07]. Service-oriented analysis and design are combined into one in [PvdH07] as *Service and Pro-*

cess Analysis and Design as the second step after planning. Analysis and design phase aims at identifying, conceptualizing and rationalizing business processes as a set of interacting Web services. The analysis and design phase is followed by Service development called the *Realization* phase in [PvdH07]. This phase takes care of the implementation, coding Web services, specified in WSDL, in any programming language; coding business processes, for instance in BPEL which is compiled and input into a BPEL execution engine. Several kinds of testing, deployment and administration follow.

As indicated in Figure 1.2, our contribution is to conceptually introduce (automated) analyses at the design and development phase before testing and deployment. For example, when the example of Figure 1.1 goes through the above mentioned development stages, special programming constructs are used. A nested scoping mechanism (called `scope`) is used to group several activities. There are also exception handling mechanisms of *fault handlers* and *compensation handlers*. The use of *compensation* allows the possibility to reverse/undo successfully completed operations, see Chapter 3 and 4 for details. All these mechanisms are part of communication and concurrency in service-based systems and thus require the use of analysis techniques to explore certain behaviours that may be difficult to handle by standard testing.

As an example, consider a scenario in which a scope named A contains two scopes named B and C, which both have compensation handlers. Assume that C has completed successfully and therefore can be compensated. However, an activity in scope B throws a fault that propagates to a fault handler in A:

$$\langle A \ldots \langle C \ldots C_{comp} \rangle ; \langle B \ldots Fault \ldots \rangle \ldots A_{Faulthandler} \rangle$$

If the A fault handler invokes compensation to undo the effect of the inner blocks, it is not done, because B has not terminated successfully. Its compensation handler is not active, thus its predecessor C cannot be compensated.

Traditional testing would hardly catch this scenario, because the fault might be thrown at runtime by a dynamically connected service. Thus, there is a clear need to employ analysis techniques at the development phase of SOA-based systems in order to improve their reliability. The effort towards providing analysis support at the development phase of SOA-based systems involves answering the following questions:

1. What languages/notations are used to represent service contracts? [OR08]

2. What are the semantics of the different language constructs? The language supports complex exception handling mechanism through compensation and fault handling constructs because one should capture

both the normal behaviours and exceptional behaviours as part of the contract specification. It has nested scoping coupled with the ability to nest concurrent activities. Dependencies among these concurrent activities are managed through *links* construct. The links include conditions that activates activities that will be performed [OOP09].

3. What is an appropriate formal model to use in analyzing service contracts? BPEL process executes concurrently and interact with remote services. In order to formally analyze BPEL orchestration, we must have a formal model of the behaviour (and a potential environment) [COR07, Oki09].

4. What are the properties to be analyzed? An example could be: Does the system progress after a compensation is made? If there is a fault during compensation, does the system continue abnormally or end in a deadlock state? [COR07, COR08]

5. What analysis techniques should be used to analyze properties? The possible techniques have different properties with respect to the computational resources they require, as well as their difference in strength when it comes to showing properties [FPO+09].

6. How do we compare different contract specifications? (operational vs logic based) [FPO+09].

7. How should one analyze the behavioural properties of services? [COR08].

8. How should the semantic models be derived? [COR07].

The issues mentioned above motivate the research on *Analysis and Verification of Service Contracts* reported in this thesis.

1.1 Problem Statement

The technical problem we address in this thesis is: specifying, analyzing and verifying the behavioural aspects of service contracts as motivated in Section 1. More precisely, based on a specific implementation of Web services orchestration in standard BPEL, formalize and analyze the behaviour of such services.

Lots of approaches abound that map BPEL to some intermediate formalism. These range from a theoretical to an operational level (details given

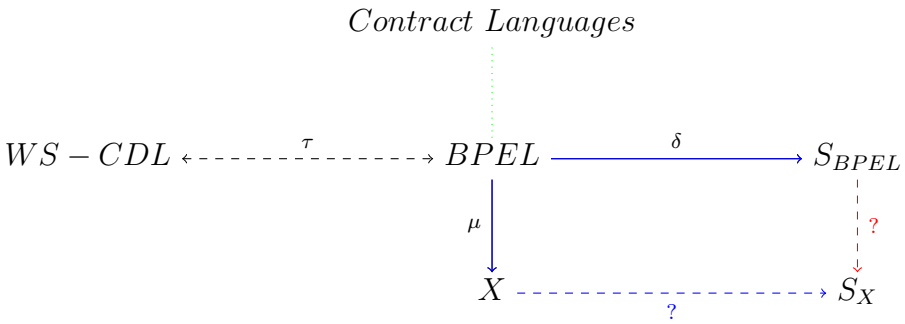


Figure 1.3: The Service Contract Analysis Problem

in 2.4) approaches. The overall observation about these efforts is that they all deal with three major issues: semantics definition, mapping to a target language and applicability. In Figure 1.3, δ represents those efforts that cover semantics definition and mostly apply Petri net simulation while μ represents those that focus on mapping/translation to another target language.

Despite all the results, the advanced features and complexity of service orchestration make it hard to fully understand and even develop a reliable high quality system. We know that analysis and verification has been successful in verifying hardware systems and safety critical software systems [EM95, CRNMB99], to cite a few examples. We therefore aim to formalize the (BPEL) service contract language constructs in an operational manner, making it amenable for analysis, thus giving a better understanding of the execution of services behaviour. This also means that analysis as well as verification can be carried out naturally. We pursue this by deriving automata models from service contracts specified in BPEL (and CDL) and then analyze/verify some properties. We also define a translation process to map the behaviour of services into timed automata which are expected to be amenable to automatic analysis and verification using the Uppaal tool. In order to achieve a semantic preserving translation/mapping, we define an operational semantics for BPEL. We implement the semantics in rewriting logic which become executable using the Maude tool. The executable semantics is then used to test the operational semantics.

1.2 Contributions

The contributions in this thesis is part of a broader project on "Contract-Oriented Software Development for Internet Services" (COSoDIS). The aim of the COSoDIS project - *Contract-Oriented Software Development for Inter-*

net Services is to develop novel approaches to implement and reason about contracts in a service oriented architecture.

The main results/contributions are summarized as follows:

1. Classification of service contract languages based on the aspects they cover. It is documented in paper [OR08], Joseph C. Okika and Anders P. Ravn. Classification of SOA Contract Specification Languages. In *Proceedings of The IEEE International Conference on Web Services (ICWS)*, pages 433-440, Los Alamitos, CA, USA, September 2008. IEEE CS Press.

2. An automated technique to check consistency between behavioural aspects of the contracts. It is documented in journal paper [COR08], Emilia Cambronero, Joseph C. Okika, and Anders P. Ravn. Consistency Checking of Web Service Contracts. *International Journal On Advances in Systems and Measurements*, 1(1):29-39, 2008.

3. A demonstration of how to use standard specifications to derive semantic models in the form of automata and identification of a need to set up a correspondence; a translation of the behaviour aspect of service contract to an automata for model checking. It is documented in paper [COR07], Emilia Cambronero, Joseph C. Okika, and Anders P. Ravn. Analyzing Web Service Contracts - An Aspect Oriented Approach. In *Proceedings of the International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM2007)*, pages 149-154. IEEE CS Press, November 2007.

4. An operational semantics of BPEL activities following the style of structural operation semantics. It is documented in Chapter 3 of this thesis.

5. A Formalization of the behavioural aspect of a component specification in a contract language CL and compares logic based specification of contracts to operational based specifications. It is documented in paper [FPO+09], Stephen Fenech, Gordon J. Pace, Joseph C. Okika, Anders P. Ravn, and Gerardo Schneider. On the specification of full contracts. *Electronic Notes Theoretical Computer Science - ENTCS*, 253(1):39-55, 2009.

6. Modelling components and component contracts using a relational calculus approach. It is documented in a book chapter [CHH+08], Zhenbang Chen, Abdel Hakim Hannousse, Dang Van Hung, Istvan Knoll, Xiaoshan Li, Yang Liu, Zhiming Liu, Qu Nan, Joseph C. Okika, Anders

P. Ravn, Volker Stolz, Lu Yang, and Naijun Zhan. Modelling with relational calculus of object and component systems-rCOS. In A. Rausch, R. Reussner, R. Mirandola, and Frantisek Plasil, editors, *The Common Component Modeling Example, volume 5153 of LNCS*, Chapter 3, pages 116–145. Springer, 2008.

7. A prototype executable semantics of BPEL based on Rewriting Logic and Maude. It is documented in chapter 4 of this thesis and in paper [OOP09], Joseph C. Okika, Olaf Owe, and Cristian Prisacariu. Operational Semantics for BPEL Complex features in Rewriting Logic. In *Proceedings of the 21st Nordic Workshop on Programming Theory, NWPT 09*, pages 95-97, Kgs. Lyngby, Denmark, 2009. Danmarks Tekniske Universitet.

8. A translation process to Uppaal timed automata for automatic verification. It is documented in Chapter 4 and in paper [Oki09], Joseph C. Okika. Analyzing Orchestration of BPEL Specified Services with Model Checking. In *Proceedings of the PhD Symposium of the 7th International Joint Conference on Service Oriented Computing (ICSOC/ServiceWave)*, pages 1-6, 2009.

1.3 Overview

In Chapter 2, we provide background to the thesis, including a brief introduction to Service Oriented Architecture and a detailed introduction to Business Process Execution Language (BPEL), the service contract language we have chosen to study. The technologies and standards used in specifying one or more aspects of service contracts are presented. These technologies include an effort to specifying the whole interoperability stack in a one-size-fits-it-all fashion, such as ebXML; a functionality specification such as OWL-S; a service level agreement specification, such as WSLA among others including the set of WS-* specifications which are still evolving. The second part of Chapter 2 provides a state-of-the-art survey in the areas of formalizations of service contracts, analysis and verification.

In Chapter 3, we present operational semantics for BPEL starting with the abstract syntax, semantic domains, operational semantic rules for the activities, including the advanced - scopes, compensation handler, fault handlers, event handlers and concurrent flow. This forms the basis for the analysis part in Chapter 4.

In Chapter 4, we present an encoding of the operational semantics based on rewriting logic and the Maude platform as indicated in Figure 1.4. Rewrit-

ing logic equations are used to model the static part of the process while (conditional) rewrite logic rules are used to model the dynamic behaviour. The result is an executable operational semantics which is used to test the semantics presented in Chapter 3.

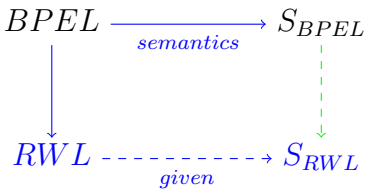


Figure 1.4: Semantics in Rewriting Logic(RWL)/Maude

In the second part of Chapter 4, we focus on giving an outline of a semantic-preserving mapping from BPEL to UppAal as indicated in Figure 1.5.

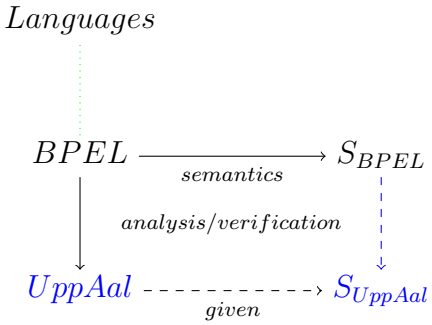


Figure 1.5: Analysis and Verification Approach

Finally, Chapter 5 concludes the thesis, presenting a summary of contributions and discusses some possible future work. Appendices include seven published papers on different aspects of analysis of services and contracts.

Chapter 2

Background and Related Work

In this chapter, we present the relevant theoretical background as well as relevant technologies including Service Oriented Architecture, Internet Services, Web Service standards, contracts and various aspects of service contracts. In addition, we present a review of the state of the art in service contract formalization, analysis and verification.

2.1 SOA and Internet Services

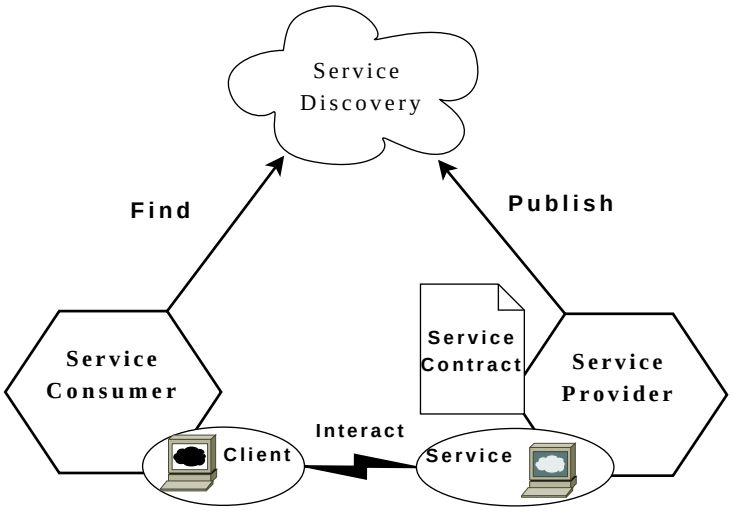
Several definitions of SOA has been given by standard bodies and in the literature. OASIS defines SOA as a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations [MLM+06].

In [Erl05], Service-Oriented Architecture (SOA) is defined as a way of reorganizing series of previously operational software applications and support infrastructure into an interconnected set of services, each accessible through standard interfaces and messaging protocols. This architectural approach is particularly applicable when multiple applications running on varied technologies and platforms need to communicate with each other. In this way, enterprises can mix and match services to perform business transactions with less programming effort. A service in this context may be defined as a behaviour that is provided by a component for use by any other component based on a network-addressable interface contract, generally identifying some capability provided by the service [RAA+05]. These services can be categorized into business services and Web services.

Business Services: A business service is a reusable combination of IT

components that delivers a business-oriented service - for instance, “Get Item Details”, to the caller, at the same time shielding the caller from any of the implementation details of that service. At a first glance, the definition of a business service seems similar to the general definition of a service given above, but the key difference is the choice of granularity of business services and the Web services.

Web Services: A Web service in the context of SOA is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols [W3C04]. They can be composed to form a higher level service or an application to solve a business goal. Services express their purpose and capabilities via a service contract as indicated in Figure 2.1 (adopted from [W3C04]). As a result of this, a detailed contractual description of services is important.



A service requestor and service provider interact based on the Service Contract

Figure 2.1: Service-Oriented Architecture.

2.2 Contracts in Computing

In real life, contracts manage social interactions (social contracts and the like), just as contracts in computing manage interaction of units of programs, components or services. Contracts are an established part of computing in

that, whenever programs, modules or units of program need to interact, some form of agreement is required. Notable background works on the issue of contracts include [BJP99, Pap03, DDK+04, Tos05].

2.2.1 Contracts for Components

As already mentioned, SOA evolved from the ideas of component based development where the notion of contract is inherent and important in developing components. This is evident from the work in the components domain: for instance, in [BJP99] the following is noted about contracts:

> Before we can trust a component in mission-critical applications, we must be able to determine, reliably and in advance, how it will behave.

The above quote can be said of services even though it is not necessarily mission-critical. Further, it poses some questions: How can you trust a component? What if the component behaves unexpectedly, either because it is faulty or simply because you misused or misunderstood the contract?

In [BJP99] components are seen to have different types of contracts which are classified into four levels. The first is the basic syntactic contract that describe the interface. The second is the behavioural contract specifying the effect of operation executions. This is followed by the synchronization contract, describing the dependencies between components, such as sequence or parallelism. The fourth level is the quality of service contract, describing requirements with respect to maximum response delay, average response, quality of the result, etc.

There are other specifications of component contracts specifications such as those based on Assumption and Guarantee in [GQ07]. At the beginning of the project upon which this thesis is based, we started by investigating a calculus for specifying components where contracts (as part of the component specification), specifies what is needed for a component to be used in building and maintaining a software without the need to know its design. This is achieved through interface specifications including the static and dynamic behaviors, interaction protocol, and tentatively timing [CHH+08].

2.2.2 Contracts for Services

Contracts are an essential part of services computing because through them, one can specify service functionality and service interactions. In this regard, a contract is a specification of the way a consuming service will interact with

the service provided. A contract for a service is defined in [Erl07] as "what establishes the terms of engagement, providing technical constraints and requirements as well as any semantic information the service owner wishes to make public". As an example, functionality-wise, service contracts specify pre-conditions which are constraints that must be satisfied before calling a service, and post-conditions, corresponding properties that are true when the call completes, and what holds about the result.

Generally, apart from the conventional legal contracts, contracts should specify both functional and non-functional requirements as well as policies that govern security procedures and access rights. These are briefly introduced below.

Business/Legal Contracts

Business contracts are mutual agreements between two or more parties engaging in various types of economic exchanges and transactions. They are used to specify the obligations, permissions and prohibitions that the signatories should be held responsible for and to state the actions or penalties that may be taken when any of the stated agreements are not being met [Gov05]. In other words, contracts are agreements on the patterns of behaviour needed to achieve mutually agreed goals, and of contingencies and sanctions to be applied if the expected behaviour is not performed. These contracts are governed by rules or laws established by the society concerned [Lin05].

Functional Requirements (as contract)

Service contracts include also a description of functional requirements, specifying what a provider will give to any consumer that chooses to abide by the terms of the contract. This may include the functions performed and the data that it will return. An earlier treatment of contracts in an object-oriented paradigm is Design by Contract [Mey97]. Here, the functional specification is achieved through assertions; which consists of preconditions, post-conditions and invariants. The framework in [Mey92] takes a pragmatic approach at code level where the assertions are part of the language. We agree that these functional specifications are important in order to specify a formal agreement between a service provider and its clients. Thus expressing what a client should do in order to make a service request and what the provider will do in return.

- Preconditions are properties that must be true when the service operation is called. It is the responsibility of the caller to guarantee that

these properties hold. If the preconditions do not hold, the operation is allowed to behave in an arbitrary manner which may lead to incorrect results or even non-termination.

- Postconditions are properties that a service operation guarantees will hold when the service operation exits. Note that if the precondition does not hold when the service operation is called, the postcondition need not hold on exit of the service operation.

- Protocols are allowed sequences of service operation calls.

Non-functional Requirements (as contract)

Service contracts can include descriptions of non-functional QoS (Quality of Service) requirements, specifying not what the service does, but how it goes about it, such as follows [CNYM00, PZ08]:

- Availability: This is concerned with whether the Web service is present or ready for immediate use.

- Accessibility: Deals with the degree of capability of serving a Web service request.

- Integrity: Integrity is about how the Web service maintains the correctness of the interaction with respect to the source.

- Performance: Performance of a Web service is measured in terms of throughput (number of Web service requests served at a given time period) and latency (round-trip time between sending a request and receiving the response).

- Reliability: This represents the degree of being capable of maintaining the service and service quality.

- Security: Security is concerned with providing confidentiality and non-repudiation by authenticating the parties involved, encrypting messages, and providing access control.

In this thesis we further categorize what goes into a service contract into different aspects.

2.2.3 Aspects of Service Contracts

In the components world, component contracts are classified into different levels as already mentioned. We follow a similar approach to define service contracts based on the aspects they cover. However, the security aspects of contracts is not covered in the component domain. Service contracts consist of aspects such as interface, functionality, Extra-functional requirements (Quality of Service), Security and behaviour [OR08].

- **Interface** defines the syntactic communication abstraction of a piece of software that is provided to an external system. It covers the type system of a particular piece of software as well as linking and marshaling; an early example is Interface Description Language (IDL) of CORBA [Vin97] or the Interface declarations in Java.

- **Functionality** refers to what functions the services provide. It is a set of operations and their specified properties that satisfy stated or implied needs. Functionality can be captured as preconditions and postconditions.

  Preconditions are properties that must be true when the service operation is called. It is the responsibility of the caller to guarantee that these properties hold. If the preconditions do not hold, the operation is allowed to behave in an arbitrary manner.

  Postconditions are properties that a service operation guarantees will hold when the service operation exits. Note that if the precondition does not hold when the service operation is called, the postcondition need not hold on exit of the service operation.

- **Security**: refers to techniques and practices that ensure confidentiality properties for a service. Security has a special treatment because it differs from other quality properties. It specifies the protocols and coding mechanisms to be used, whereas other qualities tend to give thresholds on measurable quantities.

- **Extra Functional Properties** A quality is measurable, that is: given a service, there is a function that maps it to some scale that in elaborate cases have an associated distribution function, for instance a number between 1 and 10 (with a normal distribution around 5.5). A contract on a quality gives constraints on the values acceptable for a concrete service. Examples include:

  Performance of a Web service is measured in terms of throughput (number of Web service requests served at a given time period) and latency

(round-trip time between sending a request and receiving the response).
Reliability represents the degree of being capable of maintaining the service and service quality (Mean Time between Errors) .
Availability is concerned with whether the Web service is present or ready for immediate use (Up time).
Accessibility deals with the degree of capability of serving a Web service request (Number of simultaneous users).

- **Behaviour** The behaviour of a system is a description of various scenarios of events, signals, messages, etc. The behaviour aspect of service contracts specifies synchronizations between service invocations, describing the dependencies between service operations either in sequence or in parallel. In other words, the behaviour aspect of service contracts defines the interaction protocol for collaborating services.

In the next section, we present different notations that capture one or more of the above mentioned aspects of service contracts.

2.3 Notations for Service Contracts

Several SOA standards specify or define different aspects of service contracts. We group them into three broad families; Web Services (WS-*), Semantic Web Services (*-S), and Electronic Business (eb-*). We present in detail the WSDL language, because it plays a major role in service-based systems development, and it is used or extended by some of the other languages. In addition, we illustrate with examples, one language from each of the three families which has either similar constructs for specifying contracts or cover common aspects. The intention is to give an overview of what are the set of relevant contract aspects considered by each notation, and how they are specifiied.

Web Services (WS-*)

Web Service Definition Language (WSDL) [BL06] is an interface definition language, which has an XML grammar that describes the capabilities of Web services. It serves as a (syntactic) contract between service providers and service consumers. WSDL is a processable specification which has two parts; the abstract part where interfaces and the corresponding types, messages, operations (portTypes) are specified; and the implementation or concrete part where the access point of the services are specified. In order to illustrate these points, Figure 2.2 is an example taken from [Jur06].

```
1 <?xml version="1.0" encoding="utf-8" ?>
2 <definitions xmlns:http="http://schemas.xmlsoap.org
3       /wsdl/http/"
4  xmlns:xs="http://www.w3.org/2001/XMLSchema"  ...
5  <portType name="EmployeeTravelStatusPT">
6    <operation name="EmployeeTravelStatus">
7     <input message=
8       "tns:EmployeeTravelStatusRequestMessage"/>
9     <output message=
10     "tns:EmployeeTravelStatusResponseMessage" />
11   </operation>
12 </portType>  ...
13 <message name=
14 "EmployeeTravelStatusRequestMessage">
15  <part name="employee" type="tns:EmployeeType"/>
16    </message>
17 <message name=
18 "EmployeeTravelStatusResponseMessage">
19 <part name="travelClass"
20       type="tns:TravelClassType"/>
21 </message> ...
```

The example WSDL document starts with a preamble specifying the XML version and the encoding type (line 1). This is followed by the root element `definitions` where all the namespaces used in the WSDL document are declared (line 2). Following the namespace declarations are the `portType` declarations (line 5). The `EmployeeTravelStatus` operation consists of an input and an output message (line 6). The input and output messages are also defined in WSDL as shown in line 13 and 17. Note that version 2.0 of WSDL uses *interface* for *portType* and *endpoint* for *port*.

Figure 2.2: Example WSDL document.

Business Process Modeling Notation (BPMN) The BPMN [Whi04] specification provides a graphical notation for expressing business processes in a Business Process Diagram (BPD). The objective of BPMN is to support process management by both technical users and business users by providing a notation that is intuitive to business users yet able to represent complex process semantics [Whi04]. It allows different XML-based process languages, for instance Business Process Execution Language for Web Services (BPEL), to be visualized using common elements. The BPMN specification also provides a mapping between the graphics of the notation to underlying the constructs of execution languages, particularly BPEL.

Web Services Choreography Description Language (WS-CDL) [KBR+04] allows the specification of behavioural aspects of service contracts similar to the abstract processes of BPEL. Its major

purpose is to define multi-party contracts, with the externally observable behaviour of Web services and their clients. It has an XML-language that describes a collaboration between a collection of services in order to achieve a common goal by capturing the interactions among participating services. A WS-CDL choreography description is made up of definition of activities which are performed by participants. For example, there are three types of activities in WS-CDL, *control-flow* activity, *workunit* activity and *basic* activity. Control-flow activities include, *Sequence*, *Parallel*, and *Choice*.
WSLA, WS-Policy, WS-Security, WS-Trust are some of the other languages in the Web Service family. A major quantitative aspect of a service contract is researched in [KL03], the Web Service Level Agreement (WSLA). It is targeted at defining and monitoring SLAs for Web Services. WSLA enables service customers and providers to unambiguously define the agreed performance characteristics and the way to evaluate and measure them. It has an XML-based language used by both service providers and service consumers to define parameters, metrics, service level objectives and guarantees. WSLA uses WSDL in its specification. An example *service level objective* can be specified in WSLA as depicted in Figure 2.3.

```
1 <ServiceLevelObjective name="SLO_for_AvgThroughput">
2   <soap:operation soapAction="
3           http://example.com/GetLastTradePrice"/>
4     <wsdl:input>
5          <soap:body use="literal"/>
6     </wsdl:input>
7     <wsdl:output>
8          <soap:body use="literal"/>
9     </wsdl:output>   ...
```

The example defines an obligation for the `GetLastTradePrice` operation of a Web Service with an SLA parameter `SLO_for_AvgThroughput` (average transaction throughput).

Figure 2.3: Example Service Level Objective

WS-Policy [BBC+06], specifies the policy of a Web service provider for the benefit of service consumers. In other words, WS-Policy defines a set of constructs for specifying web service policies that can be communicated to others. The specification does not define how to transport or discover a policy. Policies may be associated with various entities and resources. The policy may be associated with arbitrary XML elements and WSDL documents. The WS-PolicyAttachment specifications define such mechanisms. The policy, specified in an XML document, is transmitted to the requester using messaging specifications [BBC+06]. An example of policy definition

taken from [BGP06] is shown in Figure 2.4. It defines a policy, which must be satisfied when the credit card is about to be charged.

```
1  <wsp:Policy xml:base="http://www.bookshop.it/policies"
2        wsu:Id="BookShopPolicy"
3     xmlns:wsp="..."
4     xmlns:wsu="...">
5    <wsp:All xmlns:wsse="..."
6         xmlns:wscol="...">
7      <wsse:Confidentiality>
8        <wsse:Algorithm type="wsse:AlgSignature"
9          URI="http://www.w3.org/2000/09/xmlenc#3des-cbc"/>
10     </wsse:Confidentiality>
11     <wscol:Expression>
12      (ChargeRequest  amount) <=
13        returnInt(WSDL_XPATH, applyXPATH,
14         userpref moneyCap, up.xml)
15     </wscol:Expression>
16   </wsp:All>
17 </wsp:Policy>
```

The *BookShopPolicy* specifies a confidentiality (non-functional) property. Line 9 specifies that all exchanged messages must be encrypted using 3DES as the encryption algorithm.

Figure 2.4: Example WS-Policy definition

WS-Security [ADLH+04] is concerned with the transport of security information. For example, the information may contain a user name and password required for authentication. WS-Security standard is defined to implement security. It defines enhancements to SOAP by providing a mechanism for associating security tokens with messages. The security token may be a binary token, certificate, etc. The standard is fully extensible and can support many types of tokens. It provides support for multiple security tokens, trust domains, signature formats, and encryption technologies [ADLH+04].

WS-Trust [DLDG+02] describes a framework for trust models that enables Web services to inter-operate securely. The goal is to enable applications to construct trusted message exchanges. This trust is represented through the exchange and brokering of security tokens.

Web Service Offerings Language (WSOL) [TPP02] has an XML notation for specifying multiple classes of services for one Web service. A service offering defines one class of service for a Web Service. As classes of service for Web Services are determined by combinations of various properties, WSOL allows specification of extra-functional aspects as described in Section 2.2.3. WSOL is also compatible with WSDL.

Semantic Web Services (*-S)

OWL Web Ontology Language for Services (OWL-S) [BHL+04] emerged recently with a coverage of both functional and non-functional aspects. OWL-S is OWL ontology for semantic description of web services. The structure of OWL-S consists of a service profile for service discovery, a process model which supports composition of services, and a service grounding, which associates profile and process concepts with the underlying service interfaces. The service profile has functional and nonfunctional properties. Functional properties describe the inputs, outputs, preconditions and effects of the service (IOPEs).

The OWL-S ontology consists of four main classes that specific services should instantiate. (alternatively, service providers may create subclasses of the OWL-S classes and instantiate those instead).
Service, defines some basic concepts that tie the parts of an OWL-S service description together and holds a textual description of the service.
Profile, describes what it provides to clients, and what it requires of them. More specifically, a service profile presents the inputs, outputs, preconditions and effects of a service. This information is used for matchmaking, i.e. to find an appropriate service based on its capabilities.
Process, describes how the service works, i.e. what happens when the service is used. Services can be described as a collection of atomic or composite processes, which can be connected together in various ways, and the data and control flow can be specified.
Grounding, specifies how the service is activated, including details on communication protocols, message formats, port numbers, etc. This *abstract grounding* is usually tied to a *concrete grounding* in the form of a WSDL interface description.

The Web Service Modeling Ontology (WSMO) [Rom05] allows specification of extra-functional properties for each particular element of a Web service description. It covers a large list of such properties including; accuracy, contributor, coverage, creator, date, description, financial, format, identifier, language, network-related QoS, owner, performance, publisher, relation, reliability, rights, robustness, scalability, security, source, subject, title, transactional, trust, type, and version. At the time of writing, these properties are not included in the logical model of the Web Service Modeling Language (WSML) which is a language for expressing WSMO.

Electronic Business (eb-*)

The ebXML (electronic business XML) framework [WD06] provides a global electronic market place where enterprises of any size, anywhere, can find each other electronically and conduct business through exchange of XML based business messages. It is a standardisation effort established by the United Nations body for Trade Facilitation and Electronic Business (UN/CEFACT) and the Organisation for the Advancement of Structured Information Standards (OASIS). ebXML consists of several technologies which are provided in five main modules in the architecture. Some of these modules can be used individually and they define several aspects of service contracts.

Business Process Specification Schema (BPSS) describes collaboration between business partners, their roles, relationships and responsibilities. It defines the choreography of business documents thus covering the same domain abstractly as BPEL. The roles (of business partners) interact with each other through Business Transactions. The Business Transactions form a choreography and each Business Transaction consists of one or two document flows. An example ebXML BPSS specification of a simple notification transaction with one document flow is given in Figure 2.5. The example is adapted from the Business Process Specification Schema document [CCK+01].

```
<BusinessTransaction name="Notify of advanceshipment">
    <RequestingBusinessActivity name="">
       <DocumentEnvelope
           BusinessDocument name="ASN"/>
    </RequestingBusinessActivity>
    <RespondingBusinessActivity name=""
    </RespondingBusinessActivity>
</BusinessTransaction>
```

Figure 2.5: Example ebXML BPSS specification

ebXML BPSS defines two kinds of collaborations: binary and multi-party collaboration. Before we illustrate further, let us first introduce how the contracts between two parties are defined (CPA) and how the capabilities of a company are described (CPP).

Collaboration Protocol Profile (CPP) constrains the interaction of partners by describing the capabilities of an individual party through *Business capabilities* which describe business processes and *Technology capabilities* which describe message exchange capabilities, transport and security constraints.

Collaboration Protocol Agreement (CPA) expresses an agreement between partners. Usually, CPA is derived from CPPs of trading partners. It

describes the capabilities that trading partners have agreed to use to perform a particular business collaboration. In other words, it is a contract between two or more trading partners. CPA is also used by trading parties to set up a runtime environment for the exchange of business messages. Security characteristics of business process collaboration are also defined in BPSS, CPP and CPA as well.

As mentioned above, BPSS specifies a binary and a multi-party collaboration. A binary collaboration is always between two roles. These two roles are called *Authorized Roles*, because they represent the actors that are authorized to participate in the collaboration. The CPA/CPP specification requires that parties agree upon a Collaboration Protocol Agreement (CPA) in order to conduct a business. A CPA associates itself with a specific Binary Collaboration. A multi-party collaboration is a synthesis of binary collaborations. A multi-party collaboration consists of a number of business partner roles. Each binary pair of trading partners will be subject to one or more distinct CPAs. Each *Business Partner Role* (lines 2, 7 and 15 in Figure 2.6) performs

```
1 <MultiPartyCollaboration name="DropShip">
2   <BusinessPartnerRole name="Customer">
3    <Performs initiatingRole=
4      //binaryCollaboration[@name="Firm Order]
5    /InitiatingRole[@name=buyer]/>
6   </BusinessPartnerRole>
7   <BusinessPartnerRole name="Retailer">
8     <Performs respondingRole=
9       //binaryCollaboration[@name="Firm Order]
10     /RespondingRole[@name=seller]/>
11    <Performs initiatingRole=
12      //binaryCollaboration[@name=" Product
13   Fulfillment /InitiatingRole[@name=buyer]/>
14  </BusinessPartnerRole>
15  <BusinessPartnerRole name="DropShip Vendor">
16    <Performs respondingRole=
17      //binaryCollaboration[@name=" Product
18         Fulfillment
19       /RespondingRole[@name=seller]/>
20    </BusinessPartnerRole>
21 </MultiPartyCollaboration>
```

Figure 2.6: Example ebXML CPA specification

one *Authorized Role* in one of the binary collaborations, or perhaps one authorized role in each of several binary collaborations. This is modeled by a *Performs* element, for instance, line 3 in the example. The *Performs* linkage between a Business Partner Role and an Authorized Role is the synthesis of

Binary Collaborations into Multiparty Collaborations, line 1. Implicitly the Multiparty Collaboration consists of all the Binary Collaborations in which its Business Partner Roles play Authorized Roles [CCK+01].

BPEL - The Service Contract Language

BPEL has been designed to facilitate orchestration of web services. Numerous commercial companies were involved in the specification and standardization process leading to its successful use in several commercial applications. Also, BPEL is a programming language to specify the behaviour of interacting web services. It allows existing Web services to be orchestrated into composite services thus specifying a behavioural aspect of service contracts. BPEL uses partner link mechanisms and a number of activities to model the services interaction. Each `partnerLink` is characterized by a `partnerLinkType`, which characterizes the conversational relationship between two services by defining the roles played by each of the services in the conversation. It specifies the `portType` provided by each service to receive messages within the context of the conversation. These `portType`s are defined in a WSDL [BL06] document, and each role specifies exactly one WSDL `portType`. A WSDL document of a WS-BPEL service contains only the abstract definition of the service. The concrete part of WSDL describes the means of messaging communication technology. This is done through `partnerLinkType` elements that represent the interaction between the process service and its client services. Figure 2.7 is an example showing the declarations part of the BPEL process introduced in Figure 1.1.

Activities are categorized into two; basic and structured. Basic activities (for instance invoke, receive, etc.) define the interaction capabilities of BPEL processes whereas the structured activities are made up of constructs such as flow (for synchronization), scope, sequence, and pick activities.

Figure 2.8 is an example of the activities part of a BPEL process, containing a `flow` activity (line 2). The `flow` activity has a number of constructs, for instance `links, receive, invoke, assign`. In the example, the process waits for a client to invoke the `request` operation and stores the incoming message and parameters about the client's loan details into the `creditInformation` when started. The other constructs such as `invoke`, `assign`, etc. are used to define the business logic of checking and approving the loan based on the credit information provided by the client.

BPEL has a couple of advanced features. It has a compensation and fault handling constructs because one should capture both the normal behaviours and exceptional behaviours as part of the contract specification. Also, BPEL has nested scoping coupled with the ability to nest concurrent activities.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <bpel:process name="loanProcess" ...
3 xmlns:bpel="http://docs.oasis.org/wsbpel/2.0/.../executable">
4  <bpel:partnerLinks>
5   <bpel:partnerLink name="LoanProcessor"/></bpel:partnerLinks>
6  <bpel:variables>
7    <bpel:variable name="creditInformation"/>
8    <bpel:variable name="riskAssessment"/> </bpel:variables>
9  <bpel:faultHandlers>
10  <bpel:catch faultName="loanProcessFault" >
11   <bpel:reply faultName="loanprocess:unableToHandleRequest"
12  operation="request" partnerLink="LoanProcessor"/></bpel:catch>
13  </bpel:faultHandlers>
14  ...
15 </bpel:process>
```

The required namespaces are declared. Partner links (line 4) define different parties that interact with the BPEL process. Each partner link is related to a specific partner link type that characterizes it. The partner link in the example is named `LoanProcessor`. Since a BPEL process is itself a service, the partner link is also defined to enable interaction with the other external services. The two external services (not shown in the example) participating in this orchestration are defined as `RiskAssessment, LoanApproval`. Variables (line 6) are declared followed by fault handlers before the main body of the BPEL process, containing activities.

Figure 2.7: Example declarations in a BPEL process

```
1   ...
2  <bpel:flow>
3   <bpel:links>
4     <bpel:link name="receive-to-assess"/> </bpel:links>
5   <bpel:receive operation="request"
6     partnerLink="LoanProcessor" variable="creditInformation">
7   <bpel:reply operation="request" partnerLink="LoanProcessor"
8                   variable="approval">
9   <bpel:invoke inputVariable="creditInformation"
10      operation="check" outputVariable="riskAssessment"
11                  partnerLink="RiskAssessment">
12  <bpel:assign..../>
13 </bpel:flow>
14 </bpel:process>
```

Figure 2.8: Example activities in a BPEL process

Dependencies among these concurrent activities are managed through link constructs. The links include guard conditions. In Chapter 3, we give a more detailed discussion of these activities and their operational semantics.

2.4 Review of Related Work

In this section, we present various efforts made by researchers to bring into the services computing domain, the notion of contracts. In addition, we review several results/proposals that have formalized parts of BPEL for simulation, analysis and verification purposes. All the results/proposals are vital to understanding what BPEL is, because they all formalize one or more aspects of BPEL. As we explore these results, we focus more on a better understanding of the advanced features which are left out or simplified in some of the existing results.

Since there are lots of results in the past six years centering on formalizing service contracts (based on BPEL), we review in the second part, related work on analysis which is our major focus. An earlier overview of semantics foundation is given in [vBK06] but several new results have emerged, some of which we present below.

2.4.1 BPEL (Service Contract) Formalizations

There is a large number of papers, e.g., that formalize behaviour of Web services specified in BPEL at different levels of abstraction. Back in 2004, a formal foundation for orchestration languages with BPEL as the case study is introduced in [Vir04]. Here, the semantics of orchestration languages for Web Services is studied, defining syntax and semantics of a language to derive the interactive behaviour out of BPEL4WS specification. What makes this work interesting is that it recognized the need and paved a way to formalize the interactive behaviour of a business process. It covers the interaction activities and some basic activities such as while, assign and link operations but do not cover more complex features such as scope, and exception handling.

We could learn from their approach the semantics of activities and correlations. However, a process algebraic approach, combining a computational model with congruence rules and auxiliary operators is followed. This means that it is difficult to translate it to operational semantics.

In [ZSGX05] two specific advanced features were addressed while defining operational semantics to a simplified version of BPEL4WS. First, fault handling, where it is assumed that any fault will be caught by the fault handler of the immediately enclosing scope. Second, compensation handlers that are scope-based, fault triggered and programmable. The aim of the investigation is to use the formalization to clear up opaque points in the language and to uncover inadequate combinations or inconsistencies. In order to achieve this, a simple language (which they also call $BPEL$) is proposed, covering some of the basic activities with fault and compensation.

The work in [ZSGX05] forms a foundation toward formalizing the complex features. It will be seen later that some of the subsequent related works on the complex features use [ZSGX05] as a starting point. Big step operational semantics is considered with the environment (or data state) of the execution, mapping activities to a termination status (completes or fails); a configuration includes a process or a mark to denote the termination status, and a compensation context. A compensation context is defined as a sequence of compensation closures consisting of the name of a scope, compensation handler and a compensation context. The semantics of a process is also defined as a transition from a configuration (in an environment) to a terminated configuration. Furthermore, two forms of compensation found in BPEL are considered. They defined the compensate semantics in such a way that they are executed in reverse order. This is possible because the introduced compensation closures are accumulated in the front of the context so that a compensate command will invoke them in reverse order of their installation. In the second case, the `compensate` command looks up the compensation closure with a given name in the current compensation context. However, other advanced features such as the concurrent flow with link dependencies are not considered. Another issue is that the adopted environment and the configuration definition need to be modified if analyses should be carried out on a real BPEL program.

Next, the notion of fault, compensation and additional termination is addressed using an process algebraic approach in [ES08], where formal semantics for BPEL 2.0 fault, compensation and termination (FCT-) mechanisms are provided. This work builds upon the work in [ZSGX05] discussed above. The contribution is two-fold: to provide a detailed low-level semantics for BPEL's FCT-handling behaviour; using the semantics to place BPEL in the context of more abstract foundational work. The second point on foundational work is dealt with by comparing the BPEL presented in the paper to existing formal theories behind Sagas [GMS87]. Like the work in [ZSGX05], syntax and semantics of $BPEL_{fct}$ calculus is based on a process algebraic approach. The semantics are given in operational style together with a set of congruence rules. The notion of compensation context and compensation closures are similar to what is presented in [ZSGX05] with the introduction of a fixed compensation context. Further, the compensation closures include mode of a closure which can be normal, faulted, compensating or terminating. The essence of these two changes is to handle all-or-nothing semantics which allows repeated compensations. Unlike in [ZSGX05] the scope is defined without a name and it consists of a main activity, fault handler, compensation handler and a termination handler. Although one may say that other basic and interaction activities can easily be added to the constructs already

defined, one shortcoming in this approach is that concurrent flow construct is restricted without condition links. The construct for concurrency found in Sagas is used in defining the semantics because it is difficult to encode the flow construct with condition links in algebraic languages like the one used in [ZSGX05]. This means we must expect to define a different concurrency semantics that reflects BPEL's flow activity.

In [LPT08] a lightweight version of BPEL (called B*lite*) covering partnerlinks, termination, correlation, long running business transactions and compensation handlers is introduced and formalized. Here, a scope activity groups a primary activity together with a fault handling activity and a compensation handling activity. States are (partial) functions mapping variables to values. The operational semantics are defined in terms of structural congruence and a reduction relation. The structural congruence identifies syntactically different terms which represent the same term, defined as the least congruence relation induced by a given set of equational laws. This work does not consider concurrent flow activity with link conditions and compensation of named scopes.

There is also a large number of papers on formalizing the "simple" features (and maybe a few complex features) of BPEL. The majority of which are based on either process algebra or Petri nets. We summarize some below.

Petri-net Approaches to BPEL Formalizations

An illustrative example which is well-explained is found in [Mar05]. It deals with specification of both the abstract model and executable model of BPEL. The approach is based on Petri nets where a communication graph is generated representing a process's external visible behavior. It verifies the simulation between concrete and abstract behavior by comparing the corresponding communication graphs. Continuing with Petri net, an algebraic high-level Petri net semantics of BPEL is presented in [Sta05]. The idea here is to use the net patterns of BPEL activities in model checking certain properties of BPEL process descriptions. The model is feature complete for BPEL 1.1. Lohmann extends this work with a feature-complete net semantics for BPEL 2.0 [Loh07].

As there exists several BPEL formalizations including a comprehensive and rigorously defined mapping of BPEL constructs onto Petri net structures presented in [WVA+09, OVvdA+07] a detailed comparison and evaluation of Petri net semantics for BPEL is presented in [LVOS08]. The comparison reveals different modelling decisions with a discussion of their consequences together with an overview of the different properties that can be verified on the resulting models.

Process-Algebra Approaches to BPEL Formalizations

Abouzaid and Mullins [AM08] propose a BPEL-based semantics for a new specification language (called $BP-calculus$) based on the π-calculus, which will serve as a reverse mapping to the π-calculus based semantics introduced by Lucchi and Mazzara [LM07b]. Their mapping is implemented in a tool integrating the toolkit HAL and generating BPEL code from a specification given in the $BP-calculus$. Unlike in our approach, this work covers analysis of BPEL specifications through the mappings while the consistency of the new language and the generated BPEL code is yet to be considered. As a future work, the authors plan to investigate a two way mapping.

2.4.2 Service Contract Analysis and Verification

Back in 2004, after OASIS published BPEL4WS 1.1, some efforts [FBS04a, FBS04b, KvB04] started formalizing different parts of BPEL in order to apply different analysis techniques such as simulation and model checking. Although most of these results cover fragments of the language and some say little about the underlying analysis language and automation, they pave a way for a potential exploration of simulation and model checking techniques toward a more promising analysis framework for services based systems developed using BPEL. We discuss some of the most related approaches below.

Operational Approaches to BPEL Formalizations

In [ZK07], a formalization of BPEL based on an extension of Mealy machines (WSA) is presented. The reason for the use of extended Mealy machines is to capture the data aspect which it is claimed is abstracted from existing models. The authors noted also that the extended Mealy machine supports message passing communication and adopts interleaving semantics. Some of the complex features are covered such as flow, compensation and fault handling. The results in this work show that WSA is more general than the existing automata-based semantics in that it can model most features of BPEL and it allows verification of BPEL control and data flows. The verification can be done using NuSMV and SPIN model checkers.

One limitation in this work is that it did not say how the data is handled by the model checker. Model checking systems with generally unbounded data types are clearly not feasible. Further, WSA has no hierarchy, BPEL is highly structured and hierarchical if one considers the nested scoping mechanisms. Thus, it is not clear how the concurrent executions are captured with the parent-child machine communication adopted in [ZK07]. Moreover, com-

mon machine layout for all structured activities seems artificial for capturing all the activities.

In the case of using automata as models for formalizing BPEL, a few efforts are found in the literature; they focus on some fragments of BPEL constructs. For instance, Geguang et al. present a language μ-BPEL [PZWQ06] where a full operational semantics using a labeled transition system is defined for this language and maps its constructs to Extended Timed Automata. The language constructs are mapped to a simplified version of BPEL 1.1. In [FBS04a] a translation from BPEL to guarded automata is presented. The guarded automata are further translated into a Promela specification which is the language for the SPIN model checker.

In [MR08] a methodology is proposed for modeling and analyzing web services described in the BPEL language. It formalizes BPEL semantics using Algebra of Timed Processes (ATP). The behaviour of BPEL is modeled in discrete time transition systems. It defines a formal semantics for BPEL in terms of process algebraic rules, taking into account the discrete-timing aspects. An automated generation of models (state/transition graphs) from the BPEL specifications using an exhaustive simulation based on the formal semantics rules, implemented in the WSMod tool is discussed. Further, analysis of the resulting models by using standard verification tools for concurrent systems, such as the CADP tool is also discussed. However, among the advanced features, only the semantics of scope is defined without considering compensation which is one of the main features needed to fully define the behaviour of a BPEL process.

Compared to existing work, the work in [MR08] is based on a translation of BPEL directly into state/transition graphs, without using an intermediate language such as Promela or LOTOS. It also handles both the behavioral and the discrete-time aspects of BPEL descriptions. What we learn from this work is that it is feasible to generate analysis models from BPEL specification and utilize existing analysis tools for automatic verification. In our appraoch, we complement the above work by completing the missing parts. We use a dense (continuous) time instead of a discrete time, because we want to use a tool which verifies continuous time models. We define the semantics of the full BPEL activities.

Summary

Overall, the work presented in this thesis makes complementary contributions to those discussed above. Although most of these results cover fragments of the language and some say a little about the underlying analysis language and automation, they demonstrate a potential exploration of simulation in

the case of Petri net based models and several kinds of analysis for LTS (automata) based models. The semantics defined in the above mentioned formalisms paves a way towards underpinning the semantics of BPEL. In addition, gives a first step to building analysis tools. We are complementing the above related work with the missing parts, considering the scope, the event handlers, the fault handlers, the termination handler, the compensation handler and the other ("simple") features of BPEL in order to complete the formalization of the behaviour of a BPEL process.

Chapter 3

Operational Semantics for BPEL

As discussed in Chapter 2, formal semantics for BPEL has been under intense research because the language targets an important application area and still has ambiguity in the informal description in addition to the inherent difficulties of comprehending some complex features. These features include the nested concurrency model and the combination of communication, synchronous and asynchronous web service calls, exception handling, and shared resources. Apart from the common concurrency problems, such as deadlock or race condition, one needs to consider that BPEL supports developing applications with multiple services running in parallel and equipped with a rollback mechanism called compensation.

3.1 Some Issues with the Complex Features

A service in a BPEL orchestration does not have full knowledge of services that it will interact with in advance. This might lead to further synchronization problems. For instance, consider the following statement from the BPEL 2.0 standard document:

> Suppose two concurrent isolated scopes, S1 and S2, access a common set of variables and partner links (external to them) for read or write operations. The semantics of isolated scopes ensure that the results would be no different if all conflicting activities (read/write and write/write activities) on all shared variables and partner links were conceptually reordered so that either all such activities within S1 are completed before any in S2 or vice versa.

How can one ensure that there is no conflict? This requires the precision of formal analysis.

As already mentioned, the BPEL orchestration language includes certain advanced features such as scope, compensation handling, fault handling, and event handling activities, making the language more complex than the conventional high-level programming languages. In addition, BPEL includes a special kind of concurrency mechanism specified using a concurrent flow activity as illustrated in Figure 3.1, and another mechanism to map messages to a corresponding process instance using correlation sets.

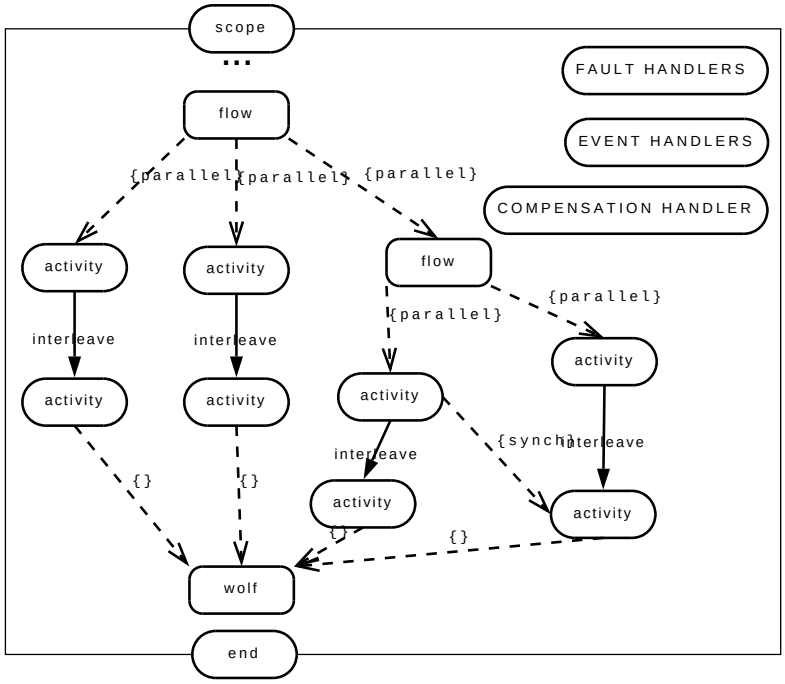


Figure 3.1: A conceptual view of BPEL complex features

We present these features in this section, starting with the scope activity.

Scope

As described in Section 12 of the standard document [AAA+07]: "A `<scope>` provides the context which influences the execution behaviour of its enclosed activities. This behavioral context includes variables, partner links, message exchanges, correlation sets, event handlers, fault handlers, a compensation handler, and a termination handler." A scope can have variables that are visible and usable at and within the scope level. Scopes can have both default

and defined fault and event handling logic, and scopes can be undone, if necessary. Undoing the work of a scope is implemented with compensation.

Flow

The flow activity handles the execution of activities in parallel together with possible dependencies. It uses synchronization links to manage dependencies, to make one activity wait for another one. Links synchronize concurrent activities as defined in the links semantics of the standard; links must have unique names in a flow within their enclosing scope. When a flow activity is started, all the directly nested activities are started concurrently and in a similar manner, a flow activity completes when all the nested activities are completed. One advantage of this could be an improvement in the response time of services.

However, concurrent execution of activities adds another level of complexity to the language, for instance, how are access to global variables, partner links and control links managed? Further, what happens to outgoing links that are never activated? Isolated scopes as mentioned in the introduction, are used to control access to global variables, partner links and control links resulting from different values due to access from different activities. This entails non-conflicting values among the different scopes. How can one analyze that there is no conflicting value based on the informal description?

Fault Handlers

A fault handler allows exception handling to be specified; it catches and handles errors within an activity. A fault handler contains an activity that will be executed when a fault occurs. When a scope receives a thrown fault message, it stops all its activities. In the default case, the scope stops its activities, and then rethrows the fault to its parent and so on until the top-level process is reached. When a scope rethrows the fault, the scope itself then ends abnormally, and activates all its outgoing links with a negative value. However, handlers allow any scope to intercept a fault and take appropriate action. Once a fault is handled in a scope, the scope is not yet terminated and proceeds with the handler activity.

Event Handler

Timed (onAlarm) events and message (onEvent) events may be introduced to define how the process deals with events that occur independent of and asynchronous to the process itself. They can be defined at either the process or scope level and they remain active as long as their enclosing scope or

process remains active. An event handler's event is triggered when a defined event occurs, either a message event or an alarm event. The message events are triggered by incoming messages, whereas alarms are triggered by either a deadline or duration.

Compensation Handler

Compensation allows (but not necessarily) undo or redo of activities performed of a scope. It is placed in a compensation handler which can only be invoked once on a successfully completed scope, from a fault, compensation or termination handler of an immediately enclosing scope. One issue that requires a close look is the link semantics defined in the BPEL standard document where it states that if the enabling condition evaluates to false then the activity is skipped and also considered completed (cf. pg 102, section 11.6 of the standard document [AAA+07]). If this assertion holds, then considering this in a scope enables it to be compensated. That means compensation is activated to attempt to reverse an activity that was never performed. It is not obvious what will happen in this situation.

Summary

Precisely identifying and formalizing the advanced features of BPEL would allow analysis that could guarantee their correct behaviour or expose some of the errors that might be hidden during development of an application. The main motive behind this work emanates from previous work on analyzing orchestration of services defined in BPEL. Here we found that it is crucial to have a complete formal semantics for BPEL including the advanced features to provide a better understanding of the entire language to facilitate developing rigorous tool supported analyses. The main contribution in this chapter of the thesis is the completion of BPEL semantics with the concurrent flow; scope with the fault handlers, event handlers, termination handler and compensation handler; the correlation sets and variable declarations. We use structural semantics since it precisely captures the order of execution of the different activities of a BPEL process and is directly usable in the analysis tool development of Chapter 4.

3.2 Abstract Syntax for Full BPEL Activities

BPEL is a sequential programming language so it has program variables, expressions and statements (called activities). Table 3.1 shows the definition of the basic terms and the syntactic sets used in the abstract syntax of BPEL

programs. A correlation set is a set of property names with specific values. For example, a correlation set name *OrderItem* may contain three properties: *ItemId*, *ItemQty* and *ItemSupplier*.

Identifiers	$x \in Var$	Set of variable names
	$p \in P$	Set of ports/partnerlinks
	$s \in S$	Set of scope names
	$f \in F$	Set of fault names
	$l \in Links$	Set of link names
	$q \in Q$	Set of property names
Expressions	$y \in Exp$	Set of expressions
	$c \in Cond \subseteq Exp$	Set of boolean conditions
	$n \in N$	Set of natural numbers
	$b \in bool$	Set of boolean values
	$d \in Time$	a time value

Table 3.1: Syntactic Categories

3.2.1 Declarations

In BPEL, partner links, variables, correlation sets are declared as part of the global declarations. A partner link represents a logical connection between the business process and a partner provided service. Variables are used to store and maintain the data of a business process. A correlation set aggregates values within a process and across several BPEL process instantiations. We do not treat the correlation set in association with process instantiations. Table 3.2 gives the abstract syntax of declarations. We use a boldface font to represent terminal symbols.

$$decl \quad ::= \quad (\textbf{partnerlink}\ p \mid \textbf{variable}\ x \mid \textbf{corset}\ q)^*$$

Table 3.2: Declarations

3.2.2 Activities

Let $\mathcal{A}_B$ denote basic activities; $\mathcal{A}_I$ denote interaction activities; $\mathcal{A}_{St}$ denote structuring activities; $\mathcal{A}_{Cc}$ denote activities that define concurrency; $\mathcal{A}_{Sc}$ and $\mathcal{A}_{comp}$ denote scope and compensate activities respectively; $\mathcal{A}$ is $\mathcal{A}_B \cup \mathcal{A}_I \cup \mathcal{A}_{St} \cup \mathcal{A}_{Cc} \cup \mathcal{A}_{Sc} \cup \mathcal{A}_{comp}$. Table 3.3 gives the abstract syntax of basic, structuring, and interaction activities.

$$
\begin{array}{lcl}
\mathcal{A} & ::= & \epsilon \mid \mathcal{A}_B \mid \mathcal{A}_I \mid \mathcal{A}_{St} \mid \mathcal{A}_{Cc} \mid \mathcal{A}_{Sc} \mid \mathcal{A}_{comp} \\
\mathcal{A}_B & ::= & \textbf{empty} \mid \textbf{exit} \mid \textbf{throw}\ f \mid \textbf{rethrow}\ f \\
 & \mid & \textbf{wait}\ d \mid \textbf{assign}\ x\ y \mid \textbf{validate}\ x \\
\mathcal{A}_I & ::= & \textbf{receive}\ p \mid \textbf{reply}\ p \mid \textbf{invoke}_S\ p_i\ p_j \mid \textbf{invoke}_A\ p \\
\mathcal{A}_{St} & ::= & \mathcal{A}\ ;\ \mathcal{A} \mid \textbf{if}\ c\ \mathcal{A}_1\ \textbf{else}\ \mathcal{A}_2 \mid \textbf{while}\ c\ \mathcal{A} \mid \textbf{repeat}\ c\ \mathcal{A} \\
 & \mid & \textbf{for}\ c\ y\ y_2\ \mathcal{A} \mid \textbf{pick}\ (p\ \mathcal{A} \quad d\ \mathcal{A})^* \\
proc & ::= & decl\ \ eventhandler^*\ \ faulthandler^*\ \ \mathcal{A}
\end{array}
$$

Table 3.3: Abstract Syntax for Basic, Interaction, and Structuring activities

3.2.3 Advanced Activities

Table 3.4 gives the abstract syntax of the advanced features. The $flow$ construct captures the concurrency aspect of BPEL. It has link declarations to manage dependencies. The other constructs are *scope* and various handlers for managing exceptions. An event handler can be a message based event (receiving a message on a specified port p) or an alarm based event (timeout of a specified time d). In the case of the fault handlers there can be a specific fault (handled by a *catch* clause in BPEL) or a set of faults (handled by a *catchAll* clause) hence the $F' \subseteq F$ in the syntax definition, where F is the set of fault names, $\mathcal{A}$ denotes the set of activities, and s a scope name.

$$
\begin{array}{lcl}
\mathcal{A}_{Cc} & ::= & \textbf{flow}\ (\textbf{link}\ l)^*\ \mathcal{A}^* \\
\mathcal{A}_{Sc} & ::= & s\ decl\ eventhandlers\ \ faulthandlers \\
 & & [compensationhandler] \\
 & & [terminationhandler]\ \mathcal{A} \\
eventhandlers & ::= & (p\ \ \mathcal{A} \mid d\ \ \mathcal{A})^* \\
faulthandlers & ::= & (F'\ \mathcal{A})^* \ \text{where}\ \ F' \subseteq F \\
compensationhandler & ::= & \mathcal{A} \\
\mathcal{A}_{comp} & ::= & \textbf{compensate} \mid \textbf{compensateScope}\ s \\
terminationhandler & ::= & \mathcal{A}
\end{array}
$$

Table 3.4: Abstract Syntax for concurrent flow, scope, and handlers

3.3 Preliminaries and Semantic Domains

The operational semantics associates to each BPEL process, an LTS whose configurations consist of the activity, a current scope, and a pair of environment and store which maps locations to values. Id is the set of names including variable names, port names, fault names, scope names:

$$p, f, s \in Id = Var \cup P \cup F \cup S$$

An environment Env maps Ids to locations, that stores values. The Env keeps track of which Ids are in scope. Thus $\rho[p]$ looks up p in environment ρ. This makes it possible for different identifiers to refer to the same location. This can be used to define handlers where the environment of activities, for instance in the same scope, are needed for subsequent executions.

$$\rho \in Env : Id \rightarrow Loc$$

We denote basic values such as integer, boolean, strings, etc., by Val. Store maps locations to basic values for simple variables, and for fault, event, compensation, and termination handlers, it keeps track of the defining scope and its environment and the associated activity. The extended structured store is needed to keep track of scope and environment in which a particular fault, event or compensation handler is to be executed.

$$\Sigma \in Store : Loc \rightarrow Val \cup (S \times Env \times \mathcal{A})$$

Thus $\sigma[l]$ is the contents of a location l in store σ.

Auxiliary Functions

We introduce an auxiliary function to capture correlation sets that map freshly declared names X to a set of properties Q (denoting name/value XML element attributes).

$$corset : X \rightarrow 2^{Q}$$

We define some auxiliary functions used in the semantics. Let $Proc$ denote a BPEL process and Env and $Store$ be as defined above.
$validate : Var \rightarrow (Env \times Store) \rightarrow bool$ - is a function that given a variable name, checks the *definededness* and type consistency.
$defined : Exp \times (Env \times Store) \rightarrow bool$ - given an expression, evaluates whether an expression/value is defined.

Evaluation of Expressions
An address function is defined to select a location:

$$addr : Id \rightarrow (Env \times Store) \rightarrow Loc$$

and an evaluation function computes the values:

$$eval : Exp \rightarrow (Env \times Store) \rightarrow Val$$

where $Id \subseteq Exp$.
Alphabet of Actions: Actions specify operations in a BPEL process. The action labels are:

- τ denotes internal actions. This action cannot be observed outside the service.
- $!p/?p$ denotes sending/receiving of messages on a specific port name. Sending of messages is denoted by $!p$ and receiving of a message is denoted by $?p$.
- $\surd$ and Ⓢ denotes termination and exit actions respectively. We distinguish between the two because successful completion is indicated by a termination in the case of a scope requiring a compensation whereas exit is used for forceful termination.
- χ denotes time elapsing. For instance `onAlarm` and `wait` activities use time elapsing actions.

System Configurations and transitions

The configuration consists of the BPEL activity yet to be executed, the name of a current scope, an environment in which the activity will be executed and a store. The operational semantics of a BPEL program is defined based on transitions between configurations, representing a snapshot of what happens during computation with *proc* representing the activity (a BPEL process) to be executed and (ρ, σ) representing the environment and values of the variables/identifiers respectively.

$$Configuration\ (proc, s, (\rho, \sigma)) \in \mathbb{C} = Proc \times S \times (Env \times Store)$$

There is an initial configuration $\mathbb{C}_0 = (proc, s_0, (\rho_0, \sigma_0))$ where $\rho_0 = [f \mapsto l_f | f \in F]$ and $\sigma_0 = [l_f \mapsto (s_0,\ \rho_0,\ implicitfh)]$ maps any fault name and mapping them to an implicit fault handler. This means any process is initially prepared to handle any fault (default fault handler in a BPEL process).

3.3.1 Declarations

A BPEL process contains a declaration part followed by the activity part. In the global declarations, the variable declarations, the partner link declarations, and the correlation sets are declared.

The variable declaration defines the data variables used by the process:

$$(\textbf{variable}\ \ x, s, (\rho, \sigma,)) \xrightarrow{\tau} (\epsilon, s, (\rho + [x \mapsto l_{new}], \sigma + [l_{new} \mapsto \bot]))$$

where a new location l_{new} is created for the variable x declared in a scope s and the value of x in the store is undefined, $\bot$.

A `partnerLink` declaration defines the different parties (port names) that interact with the process:

$$(\textbf{partnerlink}\ \ p, s, (\rho, \sigma)) \xrightarrow{\tau} (\epsilon, s, (\rho + [p \mapsto l_{new}], \sigma + [l_{new} \mapsto \bot]))$$

Similar to variable declaration, a new location l_{new} is created for the partnerlink variable p declared in a scope s and the value of p - the communication buffer in the store is undefined $\bot$.

A correlation set declaration comprised of a set of properties shared by messages. The semantics defined below, creates a new location for the name and maps it to a set of properties.

$$(\textbf{corset}\ \ x\ \ Q_a, s, (\rho, \sigma)) \xrightarrow{\tau} (\epsilon, s, (\rho + [(x \mapsto l_{x_{new}})], \sigma + [l_{x_{new_i}} \mapsto Q_a]))$$

where $Q_a \subseteq Q$ are properties.

3.3.2 Basic Activities

The semantics of basic activities are shown in Table 3.5. The *exit* rule evolves to an empty activity to depict a forceful termination. It also updates the store with, for instance, an empty compensation handler ensuring that nothing happens when a compensate command is called on the exited scope since it is not a successful completion. An alternative would be to update with a *throw* activity signaling a fault.
Note that in the *assign* and *validate* rules, f (a default fault) is thrown when the variables are not defined since there is a dedicated fault for these kinds of invalid variables and expressions in BPEL.

The *throw* activity assumes f is defined in the scope. The default is just a rethrow. $\mathcal{A}$ is an arbitrary activity. We consider both relative and

Table 3.5: Operational Semantics for BPEL - Basic Activities

Activity	Semantic Rules
empty	$(\mathbf{empty}, s, (\rho, \sigma)) \xrightarrow{\surd} (\epsilon, s, (\rho, \sigma))$
exit	$(\mathbf{exit}, s, (\rho, \sigma)) \xrightarrow{\textcircled{S}} (\epsilon, s, (\rho, \sigma'))$ where $\sigma' = \sigma + [\rho[s] \mapsto (s, \rho, \epsilon)]$
throw	$(\mathbf{throw}\ \ f, s, (\rho, \sigma)) \xrightarrow{\tau} (\mathcal{A}, s', (\rho', \sigma'))$ where $eval[f](\rho, \sigma) = (s', \rho', \mathcal{A})$ and if $s' \neq s\ \ \sigma' = \sigma + [\rho[s] \mapsto (s, \rho, \epsilon)]$ else $\sigma' = \sigma$
wait	$\dfrac{d > 0}{(\mathbf{wait}\ \ d, s, (\rho, \sigma)) \xrightarrow{\chi} (\mathbf{wait}\ (d-1), s, (\rho, \sigma))}$ $\dfrac{d \leq 0}{(\mathbf{wait}\ \ d, s, (\rho, \sigma)) \xrightarrow{\surd} (\epsilon, s, (\rho, \sigma))}$
assign	$\dfrac{defined(x)(\rho, \sigma) \wedge defined(y)(\rho, \sigma)}{(\mathbf{assign}\ x\ y, s, (\rho, \sigma)) \xrightarrow{\tau} (\epsilon, s, (\rho, \sigma + [addr[x](\rho, \sigma) \mapsto eval[y](\rho, \sigma)]))}$ $\dfrac{\neg defined(x)(\rho, \sigma) \vee \neg defined(y)(\rho, \sigma)}{(\mathbf{assign}\ x\ y, s, (\rho, \sigma)) \xrightarrow{\tau} (\mathbf{throw}\ f, s, (\rho, \sigma))}$
validate	$\dfrac{\mathbf{validate}\ x(\rho, \sigma)}{(\mathbf{validate}\ x, s, (\rho, \sigma)) \xrightarrow{\tau} (\epsilon, s, (\rho, \sigma))}$ $\dfrac{\neg\mathbf{validate}\ x(\rho, \sigma)}{(\mathbf{validate}\ x, s, (\rho, \sigma)) \xrightarrow{\tau} (\mathbf{throw}\ f, s, (\rho, \sigma))}$

absolute time for the *wait* activity. Thus we can macro expand: **wait** $a \triangleq$ **wait** $(a - now)$, indicating an absolute time wait, where a is the absolute time, *now* is the system time and $-$ is a subtraction operator.

Rethrow

The rethrow activity goes to the enclosing scope as shown by the macro expansion.

rethrow: $\mathbf{rethrow}\ \ f \triangleq \mathbf{endscope}\ ;\ \mathbf{throw}\ f$

3.3.3 Service Interaction Activities

The semantics for the **receive** activity in Table 3.6 says: in a given scope, a receive activity causes a message on port p to be received with p serving as a buffer. The **receive** activity may update the store with a received value. Similarly, the **reply** sends a message on port p.

Table 3.6: Operational Semantics for BPEL - Service Interaction Activities

Activity	Semantic Rules
receive	$(\textbf{receive}\ \ p,\ \ s, (\rho, \sigma)) \xrightarrow{?p} (\epsilon, s, (\rho, \sigma + [\rho[p] \mapsto w]))$ where w = value received $(\textbf{receive}\ \ p,\ \ s, (\rho, \sigma)) \xrightarrow{\chi} (\textbf{receive}\ \ p,\ \ s, (\rho, \sigma))$
reply	$(\textbf{reply}\ \ p,\ \ s, (\rho, \sigma)) \xrightarrow{!p} (\epsilon, s, (\rho, \sigma))$ $(\textbf{reply}\ \ p,\ \ s, (\rho, \sigma)) \xrightarrow{\chi} (\textbf{reply}\ \ p,\ \ s, (\rho, \sigma))$

Invoke

The semantic of **invoke** activity deals with firstly, asynchronous communication whereby the process executes one way invoke operation similar to **reply**; secondly, synchronous communication involving a two-way rendezvous where the process reduces to a sequence of a **reply** and **receive** activity. Both can let time pass and remain the same process.

invokeA: $\textbf{invoke}\ \ p \triangleq \textbf{reply}\ \ p$

invokeS: $\textbf{invoke}\ \ p1\ p2 \triangleq \textbf{reply}\ \ p1\ ;\ \textbf{receive}\ \ p2$

3.3.4 Structured Activities

The BPEL language provides ways to structure and nest activities.

Sequential composition

A **sequence** activity is a container that arranges and executes activities in an ordered list. This means that the first activity in a sequence executes, and when it is finished, the second activity begins. It completes when the last activity in the sequence has completed.

sequence: $$\frac{(\mathcal{A}_1, s, (\rho, \sigma)) \xrightarrow{a} (\mathcal{A}_1', s', (\rho', \sigma'))}{(\mathcal{A}_1; \mathcal{A}_2, s, (\rho, \sigma)) \xrightarrow{a} (\mathcal{A}_1'; \mathcal{A}_2, s', (\rho', \sigma'))}$$

$$(\epsilon\ ;\ \mathcal{A}, s, (\rho, \sigma)) \xrightarrow{\tau} (\mathcal{A}, s, (\rho, \sigma))$$

Conditional if

An **if** activity provides a conditional behaviour. It executes an activity based on one or more conditions defined by the **if** and optional **else-if** elements,

followed by an optional else element. The conditions are evaluated in order, and the first one to evaluate to true has its activity executed.

$$\textbf{if:}\quad \frac{eval(c)(\rho,\sigma)}{(\textbf{if}(c\ \ \mathcal{A}_1\ \ \textbf{else}\ \ \mathcal{A}_2),\ \ s,(\rho,\sigma)) \xrightarrow{\tau} (\mathcal{A}_1, s, (\rho,\sigma))}$$

$$\textbf{if:}\quad \frac{\neg eval(c)(\rho,\sigma)}{(\textbf{if}(c\ \ \mathcal{A}_1\ \ \textbf{else}\ \ \mathcal{A}_2),\ \ s,(\rho,\sigma)) \xrightarrow{\tau} (\mathcal{A}_2, s, (\rho,\sigma))}$$

$$\textbf{if}(c\ \ \mathcal{A}) \triangleq \textbf{if}(c\ \ \mathcal{A}\ \ \textbf{else}\ \ \textbf{empty})$$

$$\textbf{if}(c1\ \ \mathcal{A}_1\ \ \textbf{elseif}\ \ c2\ \ \mathcal{A}_2) \triangleq \textbf{if}(c1\ \ \mathcal{A}_1\ \ \textbf{else}\ \ (\textbf{if}\ (c2\ \ \mathcal{A}_2)))$$

while loop

The **while** activity provides for repeated execution of a contained activity. It executes an activity repeatedly until its condition evaluates to false.

$$\textbf{while:}\quad (\textbf{while}\ \ c\ \ \mathcal{A}) \triangleq \textbf{if}\ \ c\ (\mathcal{A};\ \ \textbf{while}\ \ c\ \ \mathcal{A})$$

Conditional repetition - repeatUntil

The **repeat** until activity executes an activity repeatedly until its condition evaluates to true.

$$\textbf{repeatUntil:}\quad (\textbf{repeat}\ \ \mathcal{A}\ \ c) \triangleq (\mathcal{A}; \textbf{if}\ \ c\ (\textbf{repeat}\ \ \mathcal{A}\ \ c))$$

Conditional Repetition - forEach

The **forEach** activity executes multiple instances of the same activity. This can be executed either in parallel or in sequence. We define sequential execution, since parallel execution requires a check for non-interference.

$$\textbf{forEach:}\quad (\textbf{for}\ v\ y_i\ y_f\ \mathcal{A}) \triangleq (\textbf{assign}(v, y_i); \textbf{assign}(v_2, y_f); \textbf{while}\ (v < v_2)\ \{\mathcal{A}; v := v + 1\})$$

where v_2 is a fresh variable

Selective pick

The **pick** activity waits for the occurrence of exactly one event from a set of events, then executes the activity associated with that event. After an event has been selected, the other events are no longer accepted by that `pick`. Events can be triggered in two ways: message events and alarm events. Thus, we have sets of activities: the wait set ($W = \{(wait\ d,\ \mathcal{A})...\}$) and the event set ($E = \{(p?,\ \mathcal{A})...\}$).

$$\textbf{pick 1a:}\quad \frac{(\textbf{wait}\ \ 0, \mathcal{A}) \in W}{(\textbf{pick}\ \ W\ E,\ \ s, (\rho, \sigma)) \xrightarrow{\tau} (\mathcal{A}, s, (\rho, \sigma))}$$

$$\textbf{pick 1b:}\quad \frac{\forall(\textbf{wait}\ \ d,\ \mathcal{A}) \in W : d > 0}{(\textbf{pick}\ W\ E, s, (\rho, \sigma)) \xrightarrow{\chi} (\textbf{pick}\ W'\ E, s, (\rho, \sigma)) \text{ where } W' = \{wait\ (d-1), \mathcal{A}_d \mid (wait\ d, \mathcal{A}_d) \in W\}}$$

$$\textbf{pick 2:}\quad \frac{(p?,\ \mathcal{A}) \in E}{(\textbf{pick}\ \ W\ E,\ \ s, (\rho, \sigma)) \xrightarrow{p?} (\mathcal{A}, s, (\rho, \sigma))}$$

The semantics of pick triggered by alarm (time aware) is given in rules *pick* 1*a* and *pick* 1*b*. The first rule handles the case when the specified time or duration has been reached. The second rule *pick* 1*b* handles the case when none of the timed activities have expired and thus, time can pass. Rule *pick* 2 captures the case when an event arrives.

3.3.5 Advanced Activities

As already mentioned, BPEL language has concurrency, nested scoping, and special kind of exception (and compensation) handling mechanisms.

Concurrent flow

A `flow` activity creates a set of concurrent activities nested within it. It enables synchronization dependencies between activities nested within it. This means one can define two or more activities, such as two `receive` activities, to start at the same time. The activities start when the flow starts. The flow activity completes when all the activities it contains have completed. We consider a generalized model for concurrent flow which handles both point-to-point synchronization and global synchronization.

Link Definitions

Links only apply to the flow activity. In order to define the semantics, we consider an alphabet of links as declared in BPEL, with each activity in a flow consisting of source links, target links and their corresponding link conditions), $\mathcal{A}(source\ L_s\ b_s)(target\ L_t\ b_t)$, where L_s, L_t denote source, target link names, and b_s, b_t denote the link conditions. We specifiy the following evaluation semantics for the source and the target links:

$$\textbf{source 0:}\quad \frac{\neg eval[b](\rho,\sigma)}{(source\ \ l\ b,\ \ s,(\rho,\sigma)) \xrightarrow{\tau} (\epsilon,s,(\rho,\sigma))}$$

$$\textbf{source 1:}\quad \frac{eval[b](\rho,\sigma)}{(source\ \ l\ b,\ \ s,(\rho,\sigma)) \xrightarrow{l!} (\epsilon,s,(\rho,\sigma))}$$

$$\textbf{target 0:}\quad \frac{\neg eval[b](\rho,\sigma)}{(target\ \ l\ b,\ \ s,(\rho,\sigma)) \xrightarrow{\tau} (\epsilon,s,(\rho,\sigma))}$$

$$\textbf{target 1:}\quad \frac{eval[b](\rho,\sigma)}{(target\ \ l\ b,\ \ s,(\rho,\sigma)) \xrightarrow{l?} (\epsilon,s,(\rho,\sigma))}$$

This means, if either the condition of the source link or the target link is false then ignore otherwise synchronize.
For each flow activity, we have a function:

$$alpha(\mathcal{A}_{flow}) \text{ is the set of } Links$$

Thus, links are part of the alphabet where we have unique names for source and target link names. The semantics for the flow activity is then given as follows:

$$\textbf{flow 0:}\quad \frac{A \in \mathcal{A}\quad (A,\ \ s,(\rho,\sigma)) \xrightarrow{a} (A',s,(\rho,\sigma'))}{(\textbf{flow}\ \ \mathcal{A},\ s,(\rho,\sigma)) \xrightarrow{a} (\textbf{flow}\ \ \mathcal{A}\backslash\{A\}\cup\{A'\},s,(\rho,\sigma))}\ a \notin Links,\quad A' \neq \epsilon$$

$$\textbf{flow 1:}\quad \frac{A \in \mathcal{A}\ \ ,\ \ A = \epsilon}{(\textbf{flow}\ \ \mathcal{A},\ s,(\rho,\sigma)) \xrightarrow{\tau} (\textbf{flow}\ \ \mathcal{A}\backslash\{A\},s,(\rho,\sigma))}$$

$$\textbf{flow 2:}\quad \frac{A = \oslash}{(\textbf{flow}\ \ \mathcal{A},\ s,(\rho,\sigma)) \xrightarrow{\surd} (\epsilon,s,(\rho,\sigma))}$$

$$\textbf{flow 3:}\quad \frac{A_1,A_2 \in \mathcal{A}\ \ ,\ \ (A_1,\ s,(\rho,\sigma)) \xrightarrow{l?} (A_1',\ s,(\rho,\sigma'))\ \ ,\ \ (A_2,\ s,(\rho,\sigma)) \xrightarrow{l!} (A_2',\ s,(\rho,\sigma'))}{(\textbf{flow}\ Links\ \mathcal{A},\ s,(\rho,\sigma)) \xrightarrow{\tau} (\textbf{flow}\ \ \mathcal{A}\backslash\{A_1,A_2\}\cup\{A_1',A_2'\},s,(\rho,\sigma'))}\ l \in Links$$

We use interleaving semantics for the concurrent flow, where the source links move first. An activity specified in a flow can perform an arbitrary action and can terminate as specified in rule $flow$ 0 and $flow$ 1 respectively. Rule $flow$ 2 captures the termination of a flow activity when there are no more activities to be executed. If the condition of the source link evaluates to true, and a corresponding condition of a target link evaluates to true, then the links are synchronized and the associated activities are executed as indicated in rule $flow$ 3.

Scope

The scope activity provides a context for a subset of activities. It can contain fault, event, and compensation handling for activities nested within it and can also have a set of defined variables and a correlation set. A scope can encompass a logical unit of work, making it manageable to execute, and then, if need be, reverse an activity. For example, if a customer cancels a paid travel reservation, the money must be returned and the reservation must be canceled without affecting other reservations. By enclosing activities in a scope, one can create the structure and conditions in which to manage activities as a unit.

The life cycle of a scope begins with an initialization sequence for entities defined locally within the scope: initialize variables and partner links, and install fault handlers, termination handler, and event handlers. As specified in the standard document, we consider three cases of scope completion - a normal (successful) completion where the activity within a scope completes without throwing a fault, all event handlers are disabled and the compensation handler is installed; an (unsuccessful) finish with internal fault where a fault is thrown within the scope, all other running activities and event handler instances are terminated and a matching fault handler is executed; an (unsuccessful) finish with external termination where a running scope receives a termination signal (for instance due to external fault), and all other running activities and event handler instances are terminated.

$$\text{when} \quad \mathcal{E} \neq \oslash \quad \textbf{scope} \;\; s \; decl \; \mathcal{F} \; \mathcal{E} \; \mathfrak{C} \; \mathfrak{T} \; \mathcal{A} \triangleq \textbf{scope} \;\; s \; decl \; \mathcal{F} \; \mathfrak{C} \; \mathfrak{T} \;\; \textbf{flow}(\mathcal{A} \;\; \textbf{pick} \; \mathcal{E})$$

We capture the occurrence of events by macro expanding a scope with event handlers in parallel using a **pick** activity. That means event and scope proceed in parallel.

scope 0: $(\textbf{scope} \;\; s \; decl \; \mathcal{F} \; \mathfrak{C} \; \mathfrak{T} \; \mathcal{A}, \;\; s_0, (\rho, \sigma)) \xrightarrow{\tau} (decl \; \mathcal{A} \; ; \; \mathfrak{T} \; ; \; \textbf{endscope}, s_0, (\rho', \sigma'))$

where $\rho' = \rho + [s \mapsto (l_{s_{new}})] + [f \mapsto (l_{f_{new}}) | (f, \mathcal{A}) \in \mathcal{F}]$

$\sigma' = \sigma + [l_{s_{new}} \mapsto (s, \rho, \mathfrak{C})] + [(l_{f_{new}}) \mapsto (s, \rho, \mathcal{A}) \mid (f\mathcal{A}) \in \mathcal{F}]$

scope 1: $(\textbf{endscope}, \; s, (\rho, \sigma)) \xrightarrow{\tau} (\epsilon, s', (\rho', \sigma)) \quad \textbf{where} \;\; s' = \rho[s]_1, \;\; \rho' = \rho[s]_2$

In the first rule *scope* 0, when a scope is entered, it starts by executing activity $\mathcal{A}$, installs the termination handler $\mathfrak{T}$ and **endscope** in order to cater for the successful and unsuccessful completion mentioned above. A scope obtains a new location for the outer scope, maps the current scope

to the outer scope with its environment and then map all named faults to their corresponding fault handlers. When the scope completes successfully, it defines a new environment with compensation in context and then updates the store with the compensation and fault handler mappings. **endscope** is an auxiliary activity, defining what happens at the end of scope execution. When a scope finishes executing, it will lookup and change to an outer scope with a lookup of the old environment as well. When exit is executed within a scope, the system changes to an outer scope with the store unchanged. Note that nothing happens when a compensation is called in this case. It is handled by the *exit* rule in Table 3.5.

Compensation

`compensate` is an action that calls the corresponding compensation handler. The compensation handler is defined within a scope as shown above. When a scope completes successfully, the compensation handler is installed with its associated environment. Calling the corresponding compensate action executes the activity with the compensation handler. In the case of fault or forceful termination, no compensation handler is installed. The rules for `compensate` and `compensateScope` (for compensating a named scope) are given as follows:

compensate: $(\textbf{compensate},\ \ s,(\rho,\sigma)) \xrightarrow{\tau} (\mathcal{A}\ ;\ \textbf{compensate}, s', (\rho', \sigma))$

where $\quad s' = \sigma[\rho[s]]_1,\ \ \mathcal{A} = \sigma[\rho[s]]_3,\ \ \rho' = \sigma[\rho[s]]_2$

compensateScope:

$(\textbf{compensateScope}\ \ s_c,\ \ s,(\rho,\sigma)) \xrightarrow{\tau} (\mathcal{A}\ ;\ \textbf{compensate}, s_c, (\rho', \sigma))$

where $\quad \mathcal{A} = \sigma[\rho[s_c]]_2,$ scope s_c's compensation handler

Summary

We have considered the advanced of BPEL including compensation, flow, scope, various handlers, declaration of variables, partner links and correlation sets in defining an operational semantics for BPEL. Defining the semantics of these features of BPEL language raised some issues that should be considered if the intention is to analyze orchestration based on BPEL.

The semantics of the flow construct which manages concurrency require that dependency should be considered when resources are shared. This is handled by defining a generalized flow semantics that is able to handle both point-to-point synchronization where the links can be handled in pairs of

source and target; and global synchronization where the link conditions are evaluated with a view of the overall synchronization dependencies.

A correlation set is used to associate an inbound message with a specific process instance. They are lists of message properties (names) and associated values that are applied when sending and receiving messages to ensure that inbound messages are associated with the correct executing instance of a process. Therefore we treat it as a singleton variable which are declared as part of the global declarations and can be attached as a side condition to all communication but activated only once.

Thus the semantics demonstrates that there are many sources of non-deterministic interaction between the features of BPEL. These are precisely the sore spots that analyses should discover in application programs before they are deployed. The semantics therefore offers a basis for further exploration of analysis and synthesis tools for BPEL. In the following chapter, we explore developing analysis by existing simulation and model checking tools utilizing the above semantics.

Chapter 4

Towards Analysis Tools

In this chapter, we present the use of Rewriting Logic as a semantic framework, taking an operational semantics approach and formalizing some of the complex features of BPEL. The advantage of this approach is that the semantics are executable in Maude, giving a direct support for testing and experimenting with the operational semantics defined in Chapter 3. This allows us to test BPEL process constructs using the rewrite and search tools, a very helpful and promising tool in this particular setting. Feedbacks from exploring the executable semantics are also very useful for clarifying some of the ambiguities. In Section 4.4, we describe a translation process from BPEL to UppAal. The chapter is an extended version of papers G and F [OOP09, Oki09].

4.1 Operational Semantics in Rewriting Logic

We introduce rewriting logic, syntax of equational systems, unsorted and sorted equational systems, rewriting logic semantics and Maude.

4.1.1 Rewriting Logic

Rewriting Logic [Mes92, MOM02, MR07, SRM09] is an unified model of concurrency in which several models of concurrent systems can be represented in a common framework. Rewriting logic has certain properties (such as explicit representation of concurrency) which makes it a good choice for developing a semantic framework in which different systems, models of concurrency, languages, and distributed systems can be specified and analyzed. We first

introduce the syntax of equational systems and then present a rewriting logic semantics adopted from [MR04].

Syntax of Equational Systems

We use the following notations: $f, g, h, ...$ denote function symbols; $a, b, c, ...$ denote constants; $x, y, z, ...$ denote variables. Each function symbol has an *arity*, a non-negative integer representing the number of arguments it takes. A *term* is either a variable, a constant, or an expression of the form $f(t_1, t_2, ..., t_n)$ where f is the function symbol of arity n and t_i are terms. We use $r, s, t, ...$ to denote the terms. A term u is a *subterm* of t if u is t or if t is $f(t_1, t_2, ..., t_n)$ and u is a subterm of t_i for some i. An *equation* is an expression of the form $s = t$ where s and t are terms. An *equational system* is a set of equations. An equational system E can be sorted or unsorted. An unsorted (or one-sorted) equational system means that there is only one sort in the specification, and that the equations are unconditional whereas the sorted equational system can be many-sorted or order-sorted. A many-sorted equational specification consists of a set of sorts, a set of function symbols, and equations defining the functions. Order-sorted specifications support subsorts.

Rewriting Logic Semantics

The semantics is defined in terms of the membership equational logic variant of rewriting logic. A membership equational logic (MEL) [Mes98] *signature* is a triple (K, Σ, S), with K a set of kinds, $\Sigma = \{\Sigma_{w,k}\}_{(w,k)\in K*\times K}$ a many-kinded signature and $S = \{S_k\}_{k\in K}$ a K-kinded family of disjoint sets of sorts. The kind of sort is denoted by $[s]$. A MEL Σ-algebra A contains a set A_k for each kind $k \in K$, a function $A_f : A_{k_1} \times ... \times A_{k_n} \to A_k$ for each operator $f \in \Sigma_{k_1...k_n,k}$ and a subset $A_s \subseteq A_k$ for each sort $s \in S_k$, with the meaning that the elements in sorts are well-defined, while elements without a sort are errors. We write $T_{\Sigma,k}$ and $T_\Sigma(X)_k$ to denote the set of ground Σ-terms with kind k and of Σ-terms with kind k over variables in X respectively, where $X = \{x_1 : k_1, ..., x_n : k_n\}$ is a set of kinded variables. Given a MEL signature Σ, *atomic formulae* have either the form $t = t'$ (Σ-equation) or $t : s$ (Σ-membership) with $t, t' \in T_\Sigma(X)_k$ and $s \in S_k$; and $\Sigma - sentences$ are conditional formulae of the form $(\forall X)\varphi$ if $\bigwedge_i p_i = q_i \wedge \bigwedge_j w_j : s_j$, where φ is either a Σ-equation or a Σ-membership, and all the variables in φ, p_i, q_i and w_j are in X. A MEL theory is a pair Σ, E with Σ a MEL signature and E a set of Σ-sentences.

Definition 1 *A rewriting logic specification or a (labelled) rewriting theory* $\mathcal{R}$ *is a 4-tuple* $\mathcal{R} = (\Sigma, E, \phi, R)$ *with:*

- (Σ, E) *a membership equational theory*
- $\phi : \Sigma \longrightarrow \mathbb{N}$ *a mapping assigning to each function symbol* $f \in \Sigma$ *(with, say, n arguments) a set* $\phi(f) = \{i_1, ..., ik\}$, $1 \leq i_1 < ... < i_k \leq n$ *of* frozen argument positions *under which it is forbidden to perform any rewrites; and*
- R *a set of* labelled conditional rewrite rules *of the general form*

$$r : (\forall X) t \rightarrow t' \;\; if \;\; (\bigwedge_i u_i = u_i') \wedge (\bigwedge_j vj : s_j) \wedge (\bigwedge_k w_k \rightarrow w_k')$$

where the variables appearing in all terms among those in X, terms in each rewrite or equation have the same kind, and in each membership $v_j : s_j$ the term v_j has kind $[s_j]$. Intuitively, $\mathcal{R}$ specifies a *concurrent system*, whose states are elements of the initial algebra $T_{\Sigma/E}$ specified by (Σ, E) and whose *concurrent transitions* are specified by the rules R, subject to the frozeness imposed by ϕ.

Given a rewrite theory, $\mathcal{R} = (\Sigma, E, \phi, R)$, the sentences that it proves are universally quantified rewrites of the form, $(\forall X)\ t \longrightarrow t'$, with $t, t' \in T_{\Sigma,E}(X)_k$ for some kind k, which are obtained by finite application of the following *rules of deduction*:

1. **Reflexivity.** For each $[t] \in T_\Sigma(X)$, $\dfrac{}{(\forall X)\ t \longrightarrow t}$
2. **Equality.** $\dfrac{(\forall X)\ u \longrightarrow v \quad E \vdash (\forall X)\ u = u' \quad E \vdash (\forall X)\ v = v'}{(\forall X)\ u' \longrightarrow v'}$
3. **Congruence.** For each $f : k_1 ... k_n \longrightarrow k \in \Sigma$, with $\{1, ..., n\} - \phi(f) = \{j_1, ..., j_m\}$ with $t_i \in T_\Sigma(X)_{k_i}$, $1 \leq i \leq n$, and with $t'_{j_l} \in T_\Sigma(X)_{k_{j_l}}$, $1 \leq l \leq m$, $\dfrac{(\forall X)\ t_{j_1} \longrightarrow t'_{j_1} \quad ... \quad (\forall X)\ t_{j_m} \longrightarrow t'_{j_m}}{(\forall X)\ f(t_1, ..., t_{j_1}, ..., t_{j_m}, ..., t_n) \longrightarrow f(t_1, ..., t'_{j_1}, ..., t'_{j_m}, ..., t_n)}$
4. **Replacement.** For each $\theta : X \longrightarrow T_\Sigma(Y)$ with, say, $X = \{x_1, ..., x_n\}$, and $\theta(x_l) = p_l, 1 \leq l \leq n$, and for each rewrite rule in R of the form,

$$q : (\forall X) t \rightarrow t' \;\; if \;\; (\bigwedge_i u_i = u_i') \wedge (\bigwedge_j vj : s_j) \wedge (\bigwedge_k w_k \rightarrow w_k')$$

with $Z = \{x_{j_1}, ..., x_{j_m}\}$ the set of unfrozen variables in t and t', then,

$$(\bigwedge_r (\forall Y) p_{j_r} \longrightarrow p'_{j_r})$$

$$\frac{(\bigwedge_i (\forall Y)\theta(u_i) = \theta(u'_i)) \wedge (\bigwedge_j (\forall Y)\theta(v_j) : s_j) \wedge (\bigwedge_k (\forall Y)\theta(w_k) \longrightarrow \theta(w'_k)}{(\forall Y)\theta(t) \longrightarrow \theta'(t')}$$

where for $x \in X - Z, \theta'(x) = \theta(x)$, and for $x_{j_r} \in Z, \theta'(x_{j_r}) = p'_{j_r}, 1 \leq r \leq m$.

5. **Transitivity.** $\dfrac{(\forall X)t_1 \longrightarrow t_2 \qquad (\forall X)t_2 \longrightarrow t_3}{(\forall X)t_1 \longrightarrow t_3}$

The **Reflexivity** rule says that for any state t there is an *idle transition* in which nothing changes. The **Equality** rule specifies that the states are in fact equivalence classes modulo the equations E. The **Congruence** rule is very general form of "sideways parallelism", so that each operator f can be seen as a *parallel state constructor*, allowing its nonfrozen arguments to evolve in parallel. The **Replacement** rule supports a different form of parallelism, which could be called "parallelism under ones feet", since besides writing an instance of a rule's left hand side to the corresponding right hand side instance, the state fragments in the substitution of the rule's variables can also be rewritten, provided the variables involved are not frozen. Finally, the **Transitivity** rule allows us to build longer concurrent computations by composing them sequentially [MR04].

4.1.2 Maude

Maude [Mes00, CDE+07] is a tool that supports both equational and rewriting logic computation. In rewriting logic and Maude, data and state of a system are specified as algebraic data types by means of equational specifications. New types are defined by means of the keyword `sort`; subtype relations between types by `subsort`; operations for building values of the defined types, giving the types of arguments and result by `op`. Operations may be associative (`assoc`) or commutative (`comm`) for instance. Equations that identify terms built by these operators are defined by means of the keyword `eq`. Equations are assumed to be confluent and terminating, so that we can use equations to reduce a term t to a unique, canonical form t' that is equivalent to t (they represent the same value). These specifications can be included into a *functional module*, `fmod ... endfm`. For example, Figure 4.1 shows a functional module including (in protecting mode) two modules `STRING` and `INT`. The module contains part of BPEL syntax specification. It specifies new

```
1  fmod BPEL-SYNTAX is
2     pr STRING . pr INT .
3     sorts Decl Activity Proc .
4     sorts Ident Varbl PortVar FaultVar ScopeVar .
5     sorts Source Target Link .
6     subsorts Source Target < Link .
7     op partnerlink_ : PortVar -> Decl .
8     op variable_ : Varbl  -> Decl .
9     op _&_ : Decl Decl -> Decl [assoc comm id: nodecl] .
10    ops exit noact skipa : -> Activity .
11    op emptyact : -> Activity .
12    op throw_   : FaultVar -> Activity .
13    op rethrow_ : FaultVar -> Activity .
14    op wait_    : Exp -> Activity .
15    ...
16    op _||_ : Activity Activity -> Activity [assoc comm prec 30] .
17    op flow_/_wolf : Link FlowAct -> Activity .
18    eq skipa || A = A .
19    eq A || skipa = A .
20   ...
21 endfm
```

Figure 4.1: Example functional module specifying BPEL syntax.

types for declaration, activity and process in line 3. Operations such as *exit* activity can be defined as specified in line 10. Equations are also specified in a functional module. For example we can capture that running a skip activity in parallel with another activity is the same as running that activity. This is indicated in line 18.

The dynamic behaviour of a system is specified by rewrite rules of the form $t \longrightarrow t'$, describing the local, concurrent transitions of the system. This means, when a part of a system matches the pattern t, it can be transformed into the corresponding instance of the pattern t'. Rewrite rules are included in system modules, `mod ... endm`. Rewrite rules can take the most general possible form in the variant of rewriting logic built on top of membership equational logic, that is, they can be of the form

$$t \rightarrow t' \quad if \quad (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j wj : s_j) \wedge (\bigwedge_k p_k \rightarrow q_k)$$

with no restriction on which new variables may appear in the right hand side or in the condition. Conditions in rules are formed by an associative conjunction connective /\, allowing equations (both ordinary equations `t = t'`, and matching equations `t := t'` where new variables occurring in `t` become instantiated by matching, memberships `(t : s)`, and rewrites `t => t'`) as conditions.

A module can also import another module by: `protecting`, `extending` and `including`. `protecting` cannot modify declarations in imported module while `extending` can add new constructors to existing sorts but cannot specify new equations for existing operators. `including` can add new constructors to existing sorts and new equations for existing operators.

Maude has certain features that come handy when executing a conditional rule. One of such features is *search*. The default interpreter which is used in this thesis supports search computations. The *search* command looks for all the rewrites of a given term that match a given pattern satisfying some condition. Details of example search based on the executable semantics of BPEL is given in Section **??**.

4.2 BPEL in Maude

In order to formalize BPEL, we consider the following from Chapter 3: the abstract syntax, the notion of environment and store which is used to define the configuration and the execution of the behaviour of a BPEL process as a transition from one configuration to another. BPEL processes can be simulated by running the operational semantics in Maude with the BPEL processes as input. The set of all possible executions can be explored by Maude's search command; both breadth-first search and bounded breadth-first search.

Having defined the structural operational semantics in the previous chapter, we describe in this section, the specification of the operational semantics of BPEL in rewriting logic. We define the semantics of BPEL (*proc*) in rewriting logic by specifying a rewrite theory: $\mathcal{R}_{proc} = (\Sigma_{proc}, E_{proc}, \phi_{proc}, R_{proc})$, where Σ_{proc} is BPEL's syntax and some auxiliary operators (for example, store, environment, etc), E_{proc} specifies the deterministic features of BPEL with some auxiliary semantic operations, the frozenness information ϕ_{proc} specifies what arguments can be rewritten with rewrite rules for each operator, and the rewrite logic rules R_{proc} specify the semantics of all the dynamic/concurrent features of BPEL. Since the operational semantics are such that the transitions are between configurations, we can map directly to a rewriting relation between terms representing the configurations. For example, the rule for the assign activity:

$$\frac{defined(x) \wedge defined(e)}{(assign\ x\ e, s, (\rho, \sigma)) \xrightarrow{\tau} (\epsilon, s, (\rho, \sigma + [addr[x](\rho, \sigma) \mapsto eval[y](\rho, \sigma)]))}$$

becomes a conditional rewrite logic rule of the form:

```
< assign (x, e), sv, (rho, sigma) > =>
```

`< emptyact, sv, (rho, sigma[rho(x) / eval(e, sv, rho, sigma)]) >
if defnd?(x, (rho, sigma))` .. In this way the semantic rules become (conditional) rewrite rules, where the transition in the conclusion becomes the main rewrite of the rule, and the transitions in the premises become rewrite conditions.

We organized the specification in five modules: three functional modules and two system modules. The first functional module `BPEL-SYNTAX` is a direct formalization in Maude of the syntax of BPEL. The second functional module `EVAL` defines the evaluation of arithmetic and boolean expressions using the Maude predefined functions. The third functional module `BP-ENV-STORE` models the basic rudiments of BPEL execution such as locations, values, environments and stores. The remaining two system modules `EVALUATION-EXP` and `BPEL-EXECUTION` model the semantics. The state (configuration) of a BPEL process is modeled as a multiset (tuple) of activities, name of enclosing scope, environment and store as defined in the operational semantics presented in Chapter 3.

4.2.1 Syntax

The abstract syntax of BPEL presented in the preceding chapter is defined in a module `BPEL-SYNTAX` with `sorts Ident Varbl PortVar FaultVar ScopeVar` .. It defines identifiers including variable names, port names, fault names, and scope names. We use strings to represent variable identifiers and names in BPEL. We declare them as subsort of `Ident` which means any scope name, fault name, and scope name is an identifier. We define constructors for the different identifiers which transform them to strings using the syntax: `op SV : String -> ScopeVar .  op PV : String -> PortVar .  op FV : String -> FaultVar .`

Operators for some of the auxiliaries and declarations such as variable and partner link declarations are declared.

```
op nodecl : -> Decl .
op partnerlink_ : PVar -> Decl .
op variable_ : Varbl  -> Decl .
op _&_ : Decl Decl -> Decl [assoc comm id: nodecl] .
```

Operators for the rest of the activities are declared similarly. Figure 4.2 shows a specification of some of the activities with `[ctor]` indicating that the operator is a constructor.

```
ops endscope skipa noact :        -> Activity .
op assign     : Varbl Exp         -> Activity [ctor] .
op validate_  : Varbl             -> Activity .
***  Syntax of INTERACTION activities
op receive(_,_)   : PortVar Varbl ->  Activity .
op reply(_,_)     : PortVar Varbl ->  Activity .
op invokea(_,_)   : PortVar Varbl ->  Activity .
op invokes(__,__) : PortVar Varbl PortVar Varbl ->  Activity .
op _;_  : Activity Activity -> Activity  [ctor assoc id: noact].
...
op If_Then_Else_ : BExp Activity Activity -> Activity .
op while         : BExp Activity          -> Activity .
op repeat        : Activity BExp          -> Activity .
op for           : BExp Exp Exp Activity  -> Activity .
op pick__:__   : PortVar Activity Exp Activity -> Activity .
op _||_ : Activity Activity -> Activity [assoc comm prec 30] .
eq skipa || A = A .
eq A || skipa = A .
op nolink : -> Link .
op [_&_] : Activity Link -> FlowAct .
op flow_/_wolf : Link FlowAct -> Activity .
```

Figure 4.2: Syntax of BPEL Activities

4.2.2 Semantics

In order to specify the semantics of BPEL, we need to consider how the configuration is represented. Based on the operational semantics defined in Chapter 3, BPEL uses tuples, called configurations. In this subsection, we present the components making up the tuples and some of the operations, equations and transition rules specifying the dynamic behaviour of BPEL processes.

Configurations

We use a four tuple configuration to manage the structure of execution; the first component is the process code to be executed, the second is the scope name, the third is the environment under which the code is executed, and the fourth is the store where the values of variables are kept. We specify the configurations as follows:

```
sorts Config Config2 .

op <_,_,(_,_)> : Proc ScopeVar BEnv Store -> Config [ctor] .
op {_,_,(_,_)} : Proc ScopeVar BEnv Store -> Config2 [ctor].

op surd : -> Config2 .
```

The sorts `Proc` and `ScopeVar` are as declared before. The sorts `Config` and `Config2` are declared using syntax `sorts`. We specify `Config2` to capture activity termination. This is followed by declaring operators using syntax `op`. The first two operators are constructors (`ctor`) for sort `Config` and `Config2` respectively. The `surd` operator specifies the $\surd$ action defined in the operational semantics.

Environment and Store

As indicated in the configuration, we use both the environment and store components, by binding identifiers to locations which are associated with values in the store. The environment and store are defined in the module `BP-ENV-STORE`.

```
sorts Location Identifier .
subsorts PortVar FaultVar ScopeVar Varbl < Ident .

sort BENV .
op mtE : -> BENV .

op _|>_ : Ident Location -> BEnv [prec 20] .
op __ : BENV BENV -> BENV [assoc id: mtE prec 30] .
op _(_) : BENV Ident -> Location .
op _[_/_] : BENV Location Ident -> BENV [prec 35] .

vars p p' : Ident . vars L' L'' : Location . var rho : BENV .

eq (p |> L' rho)(p') = if p == p' then L' else rho(p') fi .
eq rho [L' / p] = (p |> L') rho .
```

The environment, declared using syntax `sort BENV .` maps identifiers to locations, declared by `op _ |> _ : Ident Location -> BENV .`. There can possibly be an empty environments, declared by `op mtE : -> BENV .`. The lookup and update functions are declared as well by `op _(_) : BENV Ident -> Location .` and `op _[_ /_] : BENV Location Ident -> BENV .` respectively. This is followed by equations. For instance, `eq (p |> L' rho)(p') = if p == p' then L' else rho(p') fi .` specifies that looking up an identifier p' in an environment with p mapped to a location L', if the identifiers are the same, then it's location is L' otherwise lookup p''s location.

Similarly, the `Store` keeps the value associated to each location. The lookup function, `op _(_) : Store Location -> BValue .` returns the value assigned to a given location in a given store, if the given location exists. The update function, `op _[_<-_] : Store Location BValue -> [Store] .` updates the given store in the given location with a given value. The specification of store is give below.

```
sort BValue .
subsort Num < Value < BValue .
subsort Boolean < BValue .
op fval :  ScopeVar ---  scope name
           BENV     ---  Environment
           Activity ---  Fault handler
       ->  BValue .
sort Store .

op nothing : -> BValue .
op mtS : -> Store .
op [_,_] : Location BValue -> Store . *** [loc(5), 5]
op _|>_ : Location BValue -> Store [prec 20] .
op __ : Store Store -> Store [assoc id: mtS prec 30] .
op _(_) : Store Location -> BValue .
op _[_/_] : Store Location BValue -> Store [prec 35] .

vars L : Location . vars V' V'' : BValue .  var sigma : Store .

eq (L |> V' sigma)(L) = V' .
eq ([L,V'] sigma)(L) = V' .
eq (L |> V' sigma)(L') = if L == L' then V' else sigma(L') fi .
eq mtS[L / V'] = (L |> V') .
eq sigma [L / V'] = (L |> V') sigma .
eq sigma [L / V'] = ([L ,V'] sigma) .
eq (L |> V') sigma (L |> V'') = (L |> V') sigma .
eq mtS[L <- V'] = [L,V'] .
eq sigma[nil <- noval] = sigma .
```

Transition Rules

The execution of a BPEL process evolves by means of transitions over configurations. Thus, transition rules capturing the semantics of BPEL activities are specified in Maude by rewrite (conditional) rules. Each rule has a name to reflect the semantics of the activity it models. The `ThrowR` rule specifies the semantics of the throw activity. The throwing of a fault f evolves to a configuration below the dashed line, containing an arbitrary activity got from evaluating f with a lookup of the environment of the fault f, (which is the fault handler activity). The rule `RethrowR` expands to a sequence of `endscope` and a `throw` activity when executed. Thus going up one level to the enclosing scope since the current scope has ended.

```
crl [ThrowR] :  < throw(fv); A, sv, (rho, sigma) >
             => -------------------------------
                < A', sv', (rho', sigma') >
            if  sv'    := scopname(evalfv(fv, sv, rho, sigma)) /\
                rho'   := fhenv(evalfv(fv, sv, rho, sigma)) /\
```

```
                A'     := fhact(evalfv(fv, sv, rho, sigma)) /\
                sigma' := sigma .

rl [RethrowR] :  < rethrow(f); A, sv, (rho, sigma) >
              => ---------------------------------------------
                 < (endscope ; throw(f)); A, sv, (rho, sigma) >  .
```

As already mentioned, the operational rules in Chapter 3 are (closely) mapped to (conditional) rewrite rules. For example the rule below maps to `RethrowR` given above.

rethrow: $rethrow\ \ f \triangleq endscope\ ;\ throw\ f$

The rule `AssignR` models the assign activity. The `assign(x, e)` evolves to a configuration that contains the rest of the activity (which can be an empty activity), the same scope, the same environment and an update to the store with a new value from the evaluation of the expression e. Note the lookup of address x in the current environment and store. The transition is specified using a conditional rewrite rule which specifies that assignment can only occur (in this case) only when the variable x is defined in the current environment.

```
crl [AsgnR] :  < assign (x, e); A, sv, (rho, sigma) >
            => ------------------------------------------
               < A, sv, (rho, sigma[rho(x) / eval(e, sv, rho, sigma)]) >
            if defnd?(x, (rho, sigma)) .
```

The sequential composition of activities executes the first activity, and when it finishes, the second activity begins to execute. The sequence completes when the last activity has completed and a sequence of empty activity and another activity evolves to that activity. The sequence construct (;) are defined to be part of atomic activites. For instance, the assign activity above is composed sequentially with another activity A which may also be an empty activity.

The rule `ParFlowR` models parallel composition which allows activities to be executed concurrently.

```
crl [ParFlowR]  :  < (A || A'); R, sv, (rho, sigma) >
                => ----------------------------------------
                   < (A'' || A'''); R, sv, (rho, sigma) >
                if  < A, sv, (rho, sigma) > =>
                     < A'', sv, (rho, sigma) >  /\
                     < A', sv, (rho, sigma) >  =>
                     < A''', sv, (rho, sigma) >  /\
                     (A || A') =/= (A'' || A''') .
```

The rules `FlowR, FlowR3` models the execution of activities concurrently. In rule `FlowR`, there is no link dependencies and thus arbitrary activities can be executed in parallel. The second rule models the sequencing of activities based on their link dependencies. Thus if the link condition of a source activity evaluates to true and there is a corresponding target activity, then the two activities are executed sequentially with the source activity executed before the activity containing the target link.

```
rl [FlowR] :    < (flow (A A') wolf), sv, (rho, sigma) >
             => ------------------------------------
                < (A || A'), sv, (rho, sigma) > .

crl [FlowR3] : < (flow src(id,be) tgt(id,be) /
                 [A & src(id',be)] [A' & tgt(id,be)] wolf),sv,(rho, sigma)>
             => ------------------------------------
                < (A ; A'), sv, (rho, sigma) >
             if  id == id' /\ lcond(src(id,be)) == T .
```

The rule `ScopeR` models what happens when entering a scope. It evolves into a sequence of activities: scope declaration and associated activity $\mathcal{A}$; termination handler TH, *endscope* and updates of the environment and the store. The changes in the environment and store include a creation of new locations for the current scope and it's associated fault handlers. The store holds the compensation handler for the enclosing scope as well as the fault handler if specified.

```
crl [ScopeR] :  < ( scope{ sv . dc , h | A }); R, s0, (rho, sigma) >
             => ------------------------------------------------
                < ((dc ; A) ; (TH ; endscope)); R, s0, (rho', sigma') >
       if (rho[loc(1) / sv ][loc(2) / j ]) => rho' /\
          (sigma[loc(1) / (sv,rho,CH)][loc(2) / (sv,rho,FH)]) => sigma' .
```

Maude platform offers tool support for execution of specified semantics, using the rewrite and search facilities. In the next section, we present the experiments on testing the semantics with some test results.

4.3 Testing the Semantics

In the Maude platform, there exist facilities that are useful for testing the specified semantics such as executing the system specification through `rewrite` and `frewrite` commands. We use the rewrite facility to test the developed semantics to see whether the resulting configurations correspond to the given semantics. For example we can rewrite a synchronous invoke

activity to see if the reply has a corresponding receive and that the message involved is actually updated in the store. The resulting configuration in Figure 4.3 shows that the semantics definition for the synchronous invoke evolves to a reply and a receive with updates on the store as expected.

```
rew < invokes(PV("Supplier") VV("ShippingInfo"), PV("shippingPT")
     VV("dummy")), SV("RootScope"),( PV("Supplier") |> loc(1)
     VV("ShippingInfo") |> loc(2)  PV("shippingPT") |> loc(3)
     VV("dummy") |> loc(4), loc(1) |> 1 loc(2) |> 5 loc(3) |> 1
     loc(4) |> 0 )
    > (P1 ^ P2) .

 rewrites: 21 in 1628036047000ms cpu (0ms real) (0 rewrites/second)
 result Config: < noact,SV("RootScope"),(PV("Supplier") |> loc(1),
     loc(1) |> 1)> P1 $
     < noact,SV("RootScope"),(VV("ShippingInfo") |> loc(2)
     PV("shippingPT") |> loc(3)
     VV("dummy") |> loc(4),
     loc(4) |> (mtS(mtE(VV("ShippingInfo")))) loc(2) |> 5
     loc(3) |> 1)> P2
```

Figure 4.3: Testing Individual Activities

In a similar manner, the tests are done for other activities. Executing a scope activity evolves to a configuration with a sequential composition of the activity contained in scope (A), the termination handler (TH), and the *endscope* actvity. dc is the declaration part of the scope while R denote the rest of the activity.

```
rew < (scope sv . dc , h | A ); R, sv, (rho, sigma) > .

 rewrites: 7 in 1628036047000ms cpu (1ms real) (0 rewrites/second)
 result Config: < dc ; A ; TH ; endscope ; R, sv, (k |> loc(3)
     j |> loc(2) i |> loc(1) rho, loc(3) |> ae loc(2) |> af
     loc(1) |> ac sigma)>
```

Figure 4.4: scope Activity

The pick activity evolves to a configuration where one activity (either the event triggered or time aware) is executed. It uses the *wait* activity which we implemented as a counter to handle the alarm case.

A flow activity can evolve to a configuration where the activities are executed sequentially ($A; A'$) as shown in Figure 4.6. In this case, the source ($src(id, be)$) and target ($tgt(id, be)$) links synchronize.

```
rew < (pick {p A : d A'}), sv, (rho, sigma) > .

 rewrites: 2 in 30010ms cpu (0ms real) (0 rewrites/second)
 result Config: < A',sv,(rho,sigma)>
```

Figure 4.5: pick Activity

```
rew < (flow src(id,be) tgt(id,be) / [A & src(id,be)]
            [A' & tgt(id,be)] wolf), sv, (rho, sigma) > .

 rewrites: 2 in 1628036047000ms cpu (0ms real) (0 rewrites/second)
 result Config: < A ; A',sv,(rho,sigma)>
```

Figure 4.6: flow Activity

A throw activity evolves to a configuration that contains the fault handler (A'), in an outer scope (sv') and a new environment (rho'). These are obtained from the evaluation of a fault variable.

```
rew < throw (FV("Fault1")); skipa, SV("Scope2"), (FV("Fault1")|>loc(10)
             SV("Scope2") |> loc(20), loc(20) |> 2
                  loc(10) |> fval(sv',rho',A') ) > .

 rewrites: 14 in 1628036047000ms cpu (0ms real) (0 rewrites/second)
 result Config: < A',sv',(rho',loc(20) |> 2 loc(10) |> fval(sv', rho', A'))>
```

Figure 4.7: throw Activity

Summary

We have presented a rewriting logic specification of the BPEL semantics in the Maude platform. We have illustrated how the executable specification can be used to test the semantics of the different constructs of the language thus improving our understanding of the language and paving a way for a semantic-preserving mapping for an improved analysis of interacting services. In a previous test, we implemented a version of the semantics with a functional composition of the environment and store component. This is because the separate environment and store coupled with the complex fault value is too complex for Maude if not flattened. Using this version with flattened environment, we found out that there is a possibility of race conditions when executing three concurrent invokes inside a flow. The race conditions issue was resolved when we extended the semantics of flow activity with links where

we assign true or false value to matching source and target links. Further, the search tool exposed non-determinism in the example BPEL process, where it happens that at certain executions, the client gets different approval messages with the same request. This is due to the introduction of fault handling activity in connection with one of the invoke activities. One possible solution to some of these problems would be to restrict the use of these advanced activities but this would certainly defeat the main purpose of the BPEL language. However, analyses can be used to study the traces of execution that depict the behaviour of these advanced activities as well as unexpected behaviour which will improve our understanding of what is happening and why such things are happening.

4.4 Translating to UppAal

In this section, we outline a high level description of a translation process that maps BPEL to UppAal. We introduce some of the elements in UppAal used in the translation process such as channel, location, guard, synchronization, etc. This is followed by the mapping of (abstract) BPEL elements to UppAal elements.

4.4.1 UppAal

Timed automata (TA) are finite automata extended with a set of real-valued variables [AD94, BLL+96]. The variables model the logical clocks in a system and are initialized to zero when the system starts. TA are used for specifying and verifying real-time and concurrent systems. For instance, a concurrent system such as a BPEL process can be modelled by a finite set of timed automata running in parallel. Logical clocks can model timing in BPEL. A network of timed automata is a finite set of timed automata that run concurrently, using the same set of variables, and synchronizing on common actions. UppAal [BLL+96] is a tool suite for modelling, simulating and analyzing specifications based on the theory of timed automata extended with additional features (eg. variables over finite domains). UppAal models are usually a network of synchronizing timed automata.

Syntactically, a time automata model in UppAal is a graph with nodes called locations and directed edges called transitions. Figure 4.8 shows an example automaton represented graphically in UppAal to illustrate some of the features/elements. It has four locations (Start, S1, S2, SetupComplete) and five transitions.

Each location can be labelled with a name and invariant. An invariant

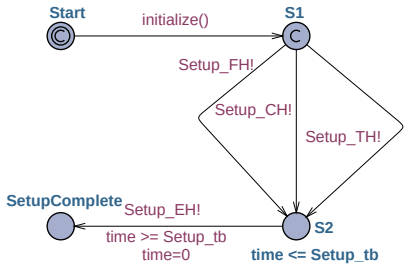


Figure 4.8: Example graphical representation of a TA in UppAal.

expresses a condition on the value of clocks and variables that must be satisfied for the automaton to stay in the location (eg. `S2` has an invariant $time <= Setup\_tb$ in Figure 4.8). A state of a TA is a pair consisting of a location and an assignment mapping all clocks and variables to values in their domains. Each TA template must have exactly one initial location, marking the beginning of a process. The initial location is marked by a double circle (eg. `Start`). Locations may be marked as committed, indicating that transitions from the location are taken atomically without interleaving with transitions of other automata in the network. In committed locations as in so called *urgent* locations, no time is passing (eg. `Start` and `S1`).

A transition may be assigned the following: a guard which may contain clock values and variables (eg. $time >= Setup\_tb$ in Figure 4.8); a synchronization through channels, where ! denotes send and ? denotes recieve (eg. `Setup_FH!` and `Setup_CH!`); and an update for updating variables and clocks (eg. $time = 0$).

4.4.2 Mapping BPEL to UppAal

A BPEL process maps to a network of timed automata and for the partnerlinks and operation attributes, we take their values as (input/output) channels. A port is mapped to a half duplex channel. A half-duplex channel allows for communication in both directions, but only one direction at a time (not simultaneously). Given a BPEL process we translate as follows:

1. Flatten any declared scope and declare channels for all specified ports,

2. Declare channels for external parallel services,

3. Declare variables for all the process variables,

4. Map each flattened scope to an automaton,

5. Map any declared event, or compensation handler to an automaton with a start and end location,

6. For each of the activities, map to a template as summarized in Table 4.1,

7. Glue associated templates and compose the system (including an environment automaton that denotes a client)

Basic activities map to UppAal templates which may contain data fields (BPEL variables). For instance, the *wait* activity translate to a template shown in Figure 4.9, consisting of three locations and two transitions with the middle location marked with an invariant $cw <= d$, meaning that the automaton may stay (wait) in the location as long as the condition is satisfied. When the condition no longer holds, the location must be left through an enabled transition. In the model of the flow activity shown in Figure 4.10, synchronization actions are used to activate enclosed activities. Similarly, updates and guards are used to keep track of active and finished activities.

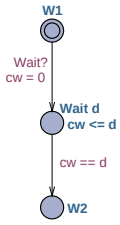


Figure 4.9: Example mapping of wait activity.

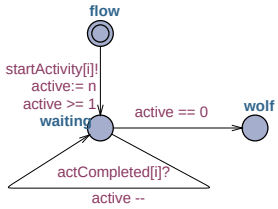


Figure 4.10: Flow

A mapping of the invoke, receive and reply activities to automata templates with value passing are shown in Figure 4.11 to Figure 4.14. The automaton with send channels (!) synchronizes with another automaton which receives (?) the sent message on the same channel. Synchronization means that two processes change location in one simulation step. A synchronization operation is done through a channel (? and !).

This translation process presents steps to extract specifications of behaviour in BPEL in the form of UppAal timed automata for analyzing behavioural properties using the UppAal tool in addition to giving semantics to BPEL. Thus composing the semantic function and the translation function, gives a property preserving semantics for BPEL based on the theory of timed automata. Hence, the semantics of BPEL consists of a collection of parallel timed automata, communicating using shared variables and synchronous communication via channels. Since the UppAal model is an abstraction,

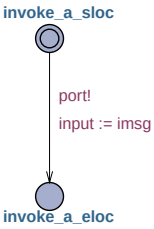


Figure 4.11: Asynchronous invoke

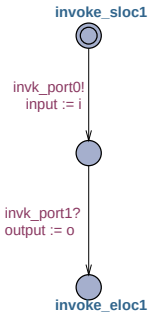


Figure 4.12: Synchronous invoke

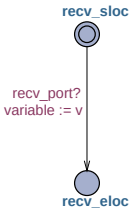


Figure 4.13: Receive

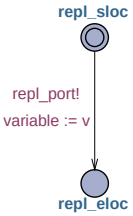


Figure 4.14: Reply

the correctness of the translation process can be established by setting up a bi-simulation relation between the UppAal models and the operational semantics presented in Chapter 3.

The next steps of the work concern implementing the translation process and refining the UppAal models of BPEL activities to make it more suitable for verification of a larger BPEL process with minimal manual modifications.

BPEL	UppAal	Remark
process	Network of TA templates	abstract all value domains
port	channel declaration	half-duplex
correlationSet	synchronization	
scope	TA template	apply flattening
fault handlers	TA template	with sloc, chan, eloc
event handlers	TA template	with sloc, chan, eloc
compensation handler	TA template	with sloc, chan, eloc
Activities		
receive	TA template	2 locs, 1 edge
reply	TA template	2 locs, 1 edge
invoke(async)	TA template	2 locs, 1 edge
invoke(sync)	TA template	3 locs, 2 edges
assign	TA template	2 locs, 1 edge
validate	TA template	2 locs, 1 edge
wait	TA template	with time guard
empty	TA template	urgent, 2 locs, 1 edge
sequence	composing transitions	depending on the activity
if	TA template	with guard, sync. action
while	TA template	with locs, inv, guard
repeatuntil	TA template	with locs, inv, guard
foreach	TA template	with locs, inv, guard
pick	TA template	with locs, inv, guard
flow	TA template	with sync. action

Table 4.1: BPEL to UppAal

Chapter 5

Conclusion

The main contributions of this thesis considering the research questions raised in the introduction are summarized as follows:

Exploring the Standards and Notations

A classification of the series of existing standard languages and notations based on the family (Web Services, Semantic Web, Electronic Business) and the aspects (functionality, quality, security, etc.) showed how and where they are used. This answers the question: what languages/notations are used to represent service contracts? The classification identifies competing languages across aspects. It shows where a language may fit into the development of service based applications as well as the ones that allow for desired analyses, for instance match of functionality, protocol compatibility or performance match. In addition, we use the classification to survey analysis approaches.

Unlike previous service contract language classification studies, we consider many aspects as well as analysis techniques. In particular, we consider families of competing languages, aspects of service contracts they cover, and formal analysis models for different classes of the languages. The classification has clear implications for service based system designers; it may assist in planning of development activities, where an application involves services with contracts that span across families. Such scenarios are to be expected as service oriented applications spread. However, the classification is not exhaustive as more standards and notations emerge while the existing standards extend with more aspects or even support tools.

Consistency Across Specifications

In [COR08] we investigated an automated technique to check consistency between protocol aspect of service contracts specified in Business Process Execution Language (BPEL) and Choreography Description Language (CDL). The contracts are abstracted to (timed) automata and from there a simulation is set up, which is checked using automated tools for analyzing simulation relations. Thus providing answers to two of the the research questions: what are the properties to be analyzed? and how should one analyze the behavioural properties of services?

We have demonstrated through a case study that, this analysis technique is applicable and gives a handle for automating yet another consistency check for web services. Unlike previous studies for instance in [DPC+05, Mar05] where either one language or one aspect of service contract is considered, we extend their perspective. Our approach considers two languages (CDL and BPEL) and two aspects of service contracts (behaviour and functionality). This is because CDL has a more detailed capture of abstract processes compared to the BPEL abstract processes. Further, BPEL is a programming language to specify the behaviour of a participant in a choreography whereas choreography is concerned with describing the message interchanges between participants. In addition, a choreography definition can be used at design time by a participant to verify that its internal processes will enable it to participate appropriately in the choreography. With this, certain properties of individual services can be verified as well as verifying the consistency between the protocols in both BPEL and CDL.

The results of the consistency checks show that the derived processes from the two service contract specifications are bisimilar and trace equivalent only when certain events (for instance fault, compensation handling in BPEL) are hidden. For example, both specifications accept the same operation sequence; since the CDL specified the protocols, while BPEL contains the operation names but with more information. They also accept the same message sequence. Thus, the state that receives the message is followed by a state that sends the message in both automata. The automaton from BPEL may contain some internal states. Therefore we note that CDL can only be consistent with an abstract version of BPEL where for example, fault handlers are hidden.

A possible future work is to streamline the tool fragments developed for these experiments, and in particular to make true the claim that the bisimulation can be integrated in an analysis process. It is well known that model checking has its limits, and investigations are also being done of theorem proving approaches [GORS06] which may be more suitable for full

implementation of conformance checking.

Analyzing Behavioural Properties

In [COR07] we analyzed behavioural properties for web service contracts formulated in Business Process Execution Language (BPEL) and Choreography Description Language (CDL), answering the question on how to analyze the behavioural properties of services. The approach includes a translation of the behavioural aspects of a service contract (in BEPL) and the functionality aspect (in CDL) to a timed automata for model checking. The approach thus covers two major aspects of contracts, and the approach lends itself to generalization to further quantitative aspects, e.g. performance analysis with queuing models. Here, performance would be analyzed with the model covering functionality, and consistency checked with the other models.

A clear difference between this approach and the related previous studies which mainly focus on either the functional aspect or the behavioural aspect of a contract is that a multi view (functional, behavioural) of a web service contract and a set of tools are proposed for the analysis while ensuring consistency. There are some potential limitations in this study. First, the timing aspects are not considered. Second the derivations are manually generated. Although the timing issue is not the main direction in this investigation, extending the models with time constraints can be naturally managed using the same tool. Thus the use of UppAal is to some extent a practical decision. We feel that it is well justified for the kinds of analyses that we discuss, because they are concerned with checking the properties of the service as such. For checking implementation conformance, it may not be ideal, and a translation to JML may be much more useful, in particular since Java may be an underlying implementation language, and JML is a formal specification language tailored to Java. Its basic use is thus the formal specification of the behaviour of Java program modules. This direction is, however, not the main line of our investigation. We have developed a translation process that maps behavioural aspects of service contracts specified in BPEL to UppAal timed automata towards providing a solution to the second issue of manual derivation. The translation process is discussed below.

Executable Semantics

We have considered an executable semantics of BPEL based on structural operational semantics, where the operational semantics associates to each BPEL process, an LTS whose configurations consist of a BPEL activity, a name of a current scope, an environment in which the activity will be executed and a store. The operational semantics of a BPEL program is then defined based on transitions between configurations.

The semantics are directly executable in the Maude platform. The Maude platform is supported with tools, which we used to test the implemented semantics. In a previous test with a functional composition of environment and store, using the search tool exposed non-determinism in an example BPEL process containing a flow activity. This is also the case with the scope and throw activity. However, there is a loss of dynamic scoping in this setting. In this case, the feedback from the tests shows that we need to separate the environment and the store and also a complex value for the fault variable. We therefore consider three component fault value; scope name, environment and fault handler (activity). The nested concurrency model of BPEL is captured in a natural way by the built-in notion of concurrency together with some rules that define interleaving. An advantage of rewriting logic is that it has a general, built-in notion of concurrency.

In Maude, concurrency is typically based on defining multisets of units (for example, activities), which become concurrent when several of these units can each be rewritten independently, meaning that each of these units changed its state (configuration). The behaviour of a system is modeled through change of the state; hence, multiple units of a system change state concurrently, i.e., multiple units in the system behave concurrently. We have used the implemented semantics to test the operational semantics of the different activities of BPEL. The feedback from the tests helps in refining the semantic definitions as well as exposing some ambiguities in the language. For example we refined the definition of the flow activity to contain a set of links and a set activities which has a corresponding source/target link. Non-determinism is inherently part of the system and this is seen when testing communication (receive, reply, invoke activities), concurrency (flow activity) and pick activity.

The implemented executable semantics gives a better understanding of the semantics of the BPEL programming language, and a suitable starting point for analyzing service contracts of both individual and connected services based on Service Oriented Architecture (SOA).

Translating to timed automata

As discussed in the related work, there are several issues around formalizing the BPEL contract language and the use of the formalizations for analysis. First, there is the issue of coverage - that is to say, how much of the language constructs (activities) are covered? Most of the efforts using automata covers some fragments of BPEL without the intricate features. A few of the efforts using Petri net covers a feature-complete BPEL. Our framework is an alternative based on automata (LTS).

Second, there is the issue of translation where one may ask: is it semantic preserving? Mapping BPEL to timed automata (TA), defines semantics for BPEL with a clear description of what is included and what is abstracted in the mapping and thus considers the issues raised above. The technique employed is systematic and can be made more rigorous by using the power of functional languages in defining a property preserving mapping for BPEL, the behaviour aspect of service contracts. It also provides an answer to the research question: what is an appropriate formal model to use in analyzing service contracts?

Following a functional approach, we could define two functions: a function that maps BPEL to Uppaal (ie. the translation process) and another function that maps UppAal to its transition system semantics (which is given). Composing these two functions relates BPEL to the given transition system semantics. Thus given semantics to BPEL. In effect, having defined the function mapping BPEL to Uppaal, we achieve a property preserving extraction/translation. Specifically, we translate as presented in Chapter 4, for example, BPEL process to a network of timed automata templates; partnerlinks and ports to channel declarations; scope and all the activities to timed automta templates. An immediate further work is the implementation of the translation process. More precisely, defining the extraction/translation function using a functional language.

Summary

As already mentioned, BPEL supports complex exception handling mechanisms through compensation and fault handling constructs because one should capture both the normal behaviours and exceptional behaviours as part of the contract specification. BPEL has nested scoping coupled with the ability to nest concurrent activities in order to support good programming styles.

Although BPEL is a complex language because of its semantics and verbose syntax, it is an industry standard for service orchestration and more

BPEL engine vendors are emerging. That means the major players in the service orchestration business will still employ it for some time in the future.

BPEL looks promising when one looks at the scope feature, supporting long running distributed transactions and the ability to do compensation. What may be inferred from these promising features is that two distinctly different programming paradigms could be considered. The first paradigm will be referred to as event handler (Event). Under this paradigm, a BPEL process can be programmed with only event handlers. This approach would still be serializable because it does not contain the intricate activities. The second paradigm will be referred to as communicating sequential process (CSP). Under this paradigm, a BPEL process can be programmed as sequential communicating processes. The two paradigms do not mix well. For instance, consider the mixture of compensation and event handlers. What happens if an event handler is a step in a transaction? It will lead to complex coding to enforce an ordering on event handlers. One can see that this scenario is better managed with sequencing of activities that run to completion. Another problem could arise due to interference in concurrent activities when event handlers and activities such as pick are combined. In this case, analysis tool may highlight these problems. However, when doing the analysis, we should consider the possibility that there is a state space explosion which is inherent in complex systems.

In conclusion, we propose that BPEL needs patterns that can be programmed by either the event handler paradigm or the communicating sequential process paradigm but not a combination of the two.

Paper A: Classification of SOA Contract Specification Languages[1]

Authors:
Joseph C. Okika
Aalborg University, Denmark

Anders P. Ravn
Aalborg University, Denmark

Abstract

There are numerous existing notations and standards in the Web service community. These may be grouped broadly into three competing families, namely; Web Services, Semantic Web, and Electronic Business. Although the families are competing, we expect that applications will cut across them and there is a need to map from one to another and to analyze compatibility and other properties. Therefore we survey how they deal with different aspects. We then illustrate with examples, the aspects of contracts captured by one language from each of the three competing families in addition to WSDL, the core standard for Web services description. The result is a classification based on the aspects of computations: functionality, protocol, and for instance performance covered by the languages. The classification is used to identify similarities between semantic models and thus find potential mappings between the families. Furthermore, this gives a handle on analysis techniques that may apply to the aspects in a particular family.

[1]This chapter is previously published in [OR08].

1 Introduction

Service oriented software systems are becoming very popular; a recent survey [Taf07] claims that adoption of SOA by enterprises will double in the next two years. Service Oriented Architecture (SOA) [Erl05] is a way of reorganizing series of previously operational software applications and support infrastructure into an interconnected services, each accessible through standard interfaces and messaging protocols. It promotes services that are distributed, heterogeneous, autonomous and open. This approach is particularly applicable when multiple applications running on varied technologies and platforms need to communicate with each other. In this way, enterprises can mix and match services to perform business transactions with less programming effort.

With increased adoption of Service Oriented Architecture and other service based frameworks, the use of service-based applications is permeating in every aspect of service computing. However, the complexity of interaction [LX05] between several service providers and service consumers leads to question of who is responsible when things go wrong. That means, there has to be a form of agreement, on the details of a service. There has to be a contract [GORS06].

Although there are several definitions and understandings of *contract*, its relevance is clear. Notable background works to the issue of contract definitions include [BJP99, Pap03, DDK+04, Tos05]. When it comes to services within SOA, we have a broader definition than the approached used in the background works. A contract is about certain *aspects* of a service such as interfaces, behaviour, functionality, and quality. A contract language expresses properties of some aspects. These properties define proper services collaborations and/or interactions.

There are many notations, languages and even standards to specify service contracts and associated techniques that support analysis of conformance for a specific service or compatibility between provided services and consuming services. We group them into three broad families; those dealing with *Web services*, those belonging to the *Semantic Web services* and those concerned with *Electronic business*. Web Service Definition Language (WSDL) [BL06] (a member of the first family), has an XML grammar that describes the capabilities of Web services through its interface descriptions. It serves as a contract between service provider and service consumer. Also in this first family are WS-BPEL [ACD+03], which covers the orchestration of services, and WS-CDL [KBR+04], which covers the choreography of interacting services. OWL-S, which describes more of functionality aspects, belongs to the second family, whereas BPSS that describes business contracts belongs to

the third family.

Our goal in this paper is to explore how these families cover key aspects of service contracts. The main contribution is a classification based on the family and the aspects. The classification identifies competing languages across aspects. It shows where a language may fit into the development of service based applications as well as the ones that allow for desired analyses, for instance match of functionality, protocol compatibility or performance match. In addition, we use the classification to survey analysis approaches. Furthermore, the classification may assist in planning of development activities, where an application involves services with contracts that span across families. Such scenarios are to be expected as service oriented applications spread.

The remaining part of the paper is organized as follows: related work is presented in Section 2. In Section 3, we introduce the different aspects of a service contracts. Section 4 presents details of notations and standards; we introduce major notations that capture service contracts through small examples. Section 5 categorizes the aspects and the notation to form the classification for service contract languages and the corresponding analysis techniques. Conclusions are given in Section 6.

2 Related work

Several comparative works exist in service contracts specification and their models. However, most papers compare pairs of languages without considering many aspects. For example, [MH05] presents a Web Service Choreography Description Language (WS-CDL) a specification for describing multiparty collaboration in conjunction with the Web Services Business Process Execution Language (WS-BPEL). It explores a mapping between the two languages. This deals only on the generation of BPEL from CDL thus covering the behavioural aspects of service contracts only. The paper [BHES07] describes transformations from Ontology Web Language-Service (OWL-S) to Business Process Execution Language (BPEL), while in [AAS06] similar work is done on developing a mapping strategy to map BPEL processes to OWL-S. The main focus of that paper is to overcome the semantic limitations of BPEL.

Some specific work covers several languages. For example, [VRO+03] presents languages such as XLANG [Tha01], BPELWS [ACD+03], ebXML [WD06], coupled with quality and security aspects. The work notes some similarities between BPML and BPEL. The work considers several languages and aspects but not analysis techniques.

The business process community uses some of the languages in developing comprehensive service compositions. For example, [LGB05] presents a survey on state of the art in modeling of Cross-Organisational Business Processes (CBP) where aspects of CBP modeling were considered based on the requirements satisfied by selected languages, which include; BPEL, CDL, and ebXML also considered here in our work. The paper [tBBG06] presents a survey on service composition approaches, where several contract languages such as BPEL and OWL-S are compared with respect to service composition requirements. Formalisms such as automata, Petri nets, and process algebra are described as composition approaches that allows for analysis.

The overall observation about the above mentioned works is that they all deal with three major issues; semantics of the languages, mappings between languages and applicability. The main difference with our work is that we consider families of competing languages and aspects they cover. Formal analysis models is also considered for different classes of the languages.

3 Service Contract Aspects

In order to give a precise, but not overly formal, meaning to the different aspects of service contracts, we will view services abstractly as a collection of Mealy machines [HU79]. Mealy machines are state machines that takes a string on an input alphabet and producing a string on an output alphabet. Outputs are a function of both the present state and the input.

Input symbols correspond to actions of a service while output symbols correspond to the results of actions. Essentially, finiteness allows for automatic analysis but since we are dealing with services, both the alphabet and the state may be infinite. For instance if we consider the actual parameters to an operation as part of the alphabet, or when we consider a very concrete specification of the state. In that case some form of abstraction is needed. The two functions, from which the next state and the value of results are determined are useful for the formulation of contracts. With this model in mind, we discuss the different aspects.

- **Interface** defines the syntactic communication abstraction of a piece of software that is provided to an external system. It covers the type system of a particular piece of software as well as linking and marshaling; an early example is Interface Description Language (IDL) of CORBA or the Interface declarations in Java. In terms of the Mealy machine, *Interface* defines the input and output alphabets.

- **Functionality** refers to what the service can do for a user. It is a

set of operations and their specified properties that satisfy stated or implied needs which for instance can be captured as preconditions and postconditions.

Preconditions are properties that must be true when the service operation is called. It is the responsibility of the caller to guarantee that these properties hold. If the preconditions do not hold, the operation is allowed to behave in an arbitrary manner. *Preconditions* are for enabling the transition in terms of the Mealy machine.

Postconditions are properties that a service operation guarantees will hold when the service operation exits. Note that if the precondition does not hold when the service operation is called, the postcondition need not hold on exit of the service operation. In terms of the Mealy machine, *Postconditions* are a specification of the next state and output.

On the whole, *Functionality* relates to the transition and output functions with precondition enabling the transition and the output function with postcondition specifying the next state and output.

- **Protocol** The behaviour of a system is a description of the input events, the response to various scenarios of events, signals, messages, etc. In the context of the Mealy machine, it can be viewed as the language accepted by the machine.

- **Security**: refers to techniques and practices that ensure confidentiality properties for a service. Security has a special treatment because it differs from other quality properties. It specifies the protocols and coding mechanisms to be used, whereas other qualities tend to give thresholds on measurable quantities.

- **Extra Functional Properties** A quality is measurable, that is: given a service, there is a function that maps it to some scale, for instance a number between 1 and 10. A contract on a quality gives constraints on the values acceptable for a concrete service. Examples include:

  Performance of a Web service is measured in terms of throughput (number of Web service requests served at a given time period) and latency (round-trip time between sending a request and receiving the response).
  Reliability represents the degree of being capable of maintaining the service and service quality.
  Availability is concerned with whether the Web service is present or ready for immediate use.

Accessibility deals with the degree of capability of serving a Web service request.

In the context of the Mealy machine, the above mentioned extra-functional properties can be handled by introducing additional functions for instance cost functions on Mealy machine or by extending Mealy machines to stochastic processes, for instance Markov processes [KS76].

In the next section, we present the different notations and standards that capture one or more of the above mentioned aspects of service contracts.

4 Service Contract Descriptions

Several SOA standards or other notations provide a way to describe/specify aspects. We group them into three broad families; Web Services (WS-*), Semantic Web Services (*-S), and Electronic Business (eb-*). We present in detail the WSDL language, because it plays a major role in SOA development, and it is used or extended by some of the other languages. In addition, we illustrate with examples, one language from each of the three families which has either similar constructs for specifying contracts or cover common aspects. The intention is to give an overview of what are the set of relevant contract aspects considered by each notation, how they are modeled and how they are possibly analyzed.

Web Services (WS-*)

We consider in this subsection, WSDL, WSOL, WS-BPEL, WS-CDL, WS-Security, WSLA, WS-Policy, and WS-Trust.

Web Service Definition Language (WSDL) [BL06], has an XML grammar that describes the capabilities of Web services through its interface. It serves as a (syntactic) contract between service providers and service consumers. WSDL is a machine-processable specification of Web service interfaces which has two parts; the abstract part where interfaces and the corresponding types, messages, operations (portTypes) are specified; and the implementation or concrete part where the access point of the services are specified. In order to illustrate these points, below is an example taken from [Jur06].

```
<?xml version="1.0" encoding="utf-8" ?>
<definitions xmlns:http="http://schemas.xmlsoap.org
      /wsdl/http/"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"  ...
 <portType name="EmployeeTravelStatusPT">
   <operation name="EmployeeTravelStatus">
```

```
    <input message=
      "tns:EmployeeTravelStatusRequestMessage"/>
    <output message=
     "tns:EmployeeTravelStatusResponseMessage" />
   </operation>
</portType>  ...
<message name=
"EmployeeTravelStatusRequestMessage">
 <part name="employee" type="tns:EmployeeType"/>
   </message>
<message name=
"EmployeeTravelStatusResponseMessage">
<part name="travelClass"
      type="tns:TravelClassType"/>
</message> ...
```

In the above example, the WSDL document starts with a preamble specifying the XML version and the encoding type. This is followed by the root element `definitions` where all the namespaces used in the WSDL document are declared. Following the namespace declarations are the `portType` declarations. The `EmployeeTravelStatus` operation consists of an input and an output message. The input and output messages are also defined in WSDL as shown in the last part of the example. Note that version 2.0 of WSDL uses *interface* for *portType* and *endpoint* for *port*.

The Business Process Execution Language for Web Services (BPEL) specifies behavioural aspects of service contracts. It uses the partner links mechanism and a number of activities to model the services interaction. Each `partnerLink` is characterized by a `partnerLinkType`, which characterizes the conversational relationship between two services by defining the roles played by each of the services in the conversation. It specifies the `portType` provided by each service to receive messages within the context of the conversation. These `portType`s are defined in the WSDL document, and each role specifies exactly one WSDL `portType`. A WSDL document of a WS-BPEL process service contains only the abstract definition of the service. The concrete part of WSDL describes the means of messaging communication technology. This is done through `partnerLinkType` sections that represent the interaction between the process service and its client services.

Activities are categorized into two; basic and structured. Basic activities (for instance invoke, receive, etc.) define the interaction capabilities of BPEL processes whereas the structured activities are made up of constructs such as flow (for synchronization), switch, and pick in addition to the basic activities. For example:

```
<?xml version="1.0" encoding="utf-8" ?>
<process name="Travel"
 xmlns="http://schemas.xmlsoap.org/ws/2003/03/
  business-process/"
 ...
 <partnerLinks>
```

```
   <partnerLink name="client"
      partnerLinkType="trv:travelLT"
       myRole="travelService"/> ...
   <partnerLink name="employeeTravelStatus"
      partnerLinkType="emp:employeeLT"
   partnerRole="employeeTravelStatusService"/>
      ...
   <partnerLink name="AmericanAirlines"
      partnerLinkType="aln:flightLT"
       myRole="airlineCustomer"
    partnerRole="airlineService"/>
   <partnerLink name="DeltaAirlines"
      partnerLinkType="aln:flightLT"
       myRole="airlineCustomer"
    partnerRole="airlineService"/>
 </partnerLinks>
<!-- Variables are declared here-->
 <sequence>
   <receive partnerLink="client"
     portType="trv:TravelApprovalPT"
       operation="TravelApproval"
       variable="TravelRequest"
    createInstance="yes" />
    ...
```

The required namespaces are declared. Partner links define different parties that interact with the BPEL process. Each partner link is related to a specific partner link type that characterizes it. The example shows four roles. The first partner link is called `client` and it is characterized by the `travelLT` partner link type. In order to enable the client to invoke the business process, we need to specify the `myRole` attribute to describe the role of the BPEL process, that is, `travelService`. The second partner link `employeeTravelStatus` is characterized by the `emp:employeeLT` partner link type. It is a synchronous request/response relation between the BPEL process and the web service; we again specify only one role. This time it is the partner role, because we describe the role of the web service, which is a partner to the BPEL process.

The last two partner links correspond to the airlines web services. Because they use the same type of web service, two partner links based on a single partner link type, `aln:flightLT` is specified. Here we have asynchronous callback communication, therefore we need two roles. The role of the BPEL process `myRole` to the airline web service is `airlineCustomer`, while the role of the airline `partnerRole` is `airlineService`. Variables are declared as indicated in the example before the main body of a BPEL process which contains a `sequence` activity. The `sequence` activity has a number of constructs, for instance `receive`. In the example the process waits for the client to invoke the `TravelApproval` operation and stores the incoming message and parameters about the business trip into the `TravelRequest`. Other constructs such as invoke, assign, etc are declared in a similar manner.

Web Services Choreography Description Language (WS-

CDL) [KBR+04] allows the specification of the behavioural aspect of service contracts similar to the abstract processes of BPEL. Its major purpose is to define multi-party contracts, which describe the externally observable behavior of web services and their clients. It has an XML-language that describes a collaboration between a collection of services in order to achieve a common goal by capturing the interactions among participating services. A WS-CDL choreography description is made up of definition of activities which are performed by participants. For example, there are three types of activity in WS-CDL, *control-flow* activity, *workunit* activity and *basic* activity. Control-flow activities include, Sequence, Parallel, and Choice.

WSLA, WS-Policy, WS-Security, WS-Trust are some of the other languages in the Web Service family. A major quantitative aspect of a service contract is researched in [KL03]. The Web Service Level Agreement (WSLA) framework [KL03] is targeted at defining and monitoring SLAs for Web Services. WSLA enables service customers and providers to unambiguously define the agreed performance characteristics and the way to evaluate and measure them. It has an XML-based language used by both service providers and service consumers to define parameters, metrics, service level objectives and guarantees. WSLA references WSDL in its specification. An example *service level objective* can be specified in WSLA as follows:

```
<ServiceLevelObjective name="SLO_for_AvgThroughput">
   <soap:operation soapAction="
           http://example.com/GetLastTradePrice"/>
     <wsdl:input>
          <soap:body use="literal"/>
     </wsdl:input>
     <wsdl:output>
          <soap:body use="literal"/>
     </wsdl:output>   ...
```

WS-Policy [BBC+06], specifies the policy of a web service provider for the benefit of service consumers. In other words, WS-Policy defines a set of constructs for specifying web service policies that can be communicated to others. The specification does not define how to transport or discover a policy. Policies may be associated with various entities and resources. The policy may be associated with arbitrary XML elements and WSDL documents. The WS-PolicyAttachment specifications define such mechanisms. The policy, specified in an XML document, is transmitted to the requester using messaging specifications [BBC+06].

WS-Security [ADLH+04] is concerned with the transport of security information. For example, the information may contain a user name and password required for authentication. WS-Security standard is defined to implement security. It defines enhancements to SOAP by providing a mechanism for associating security tokens with messages. The security token may be a binary token, certificate, etc. The standard is fully extensible and can support

many types of tokens. It provides support for multiple security tokens, trust domains, signature formats, and encryption technologies [ADLH+04].

WS-Trust [DLDG+02] describes a framework for trust models that enables Web services to securely inter-operate. The goal is to enable applications to construct trusted message exchanges. This trust is represented through the exchange and brokering of security tokens.Web Service Offerings Language (WSOL) [TPP02] has an XML notation for specifying multiple classes of services for one Web service. A service offering defines one class of service for a Web Service. As classes of service for Web Services are determined by combinations of various properties, WSOL allows specification of extra-functional aspects as described in Section 3. WSOL is also compatible with WSDL.

Semantic Web Services (*-S)

OWL Web Ontology Language for Services (OWL-S) [BHL+04] emerged recently with a coverage of both functional and non-functional aspects. OWL-S is OWL ontology for semantic description of the web services. The structure of OWL-S consists of a service profile for service discovery, a process model which supports composition of services, and a service grounding, which associate profile and process concepts with the underlying service interfaces. The Service profile has functional and nonfunctional properties. Functional properties describe the inputs, outputs, preconditions and effects of the service (IOPEs).

The OWL-S ontology consists of four main classes that specific services should be instantiated. (Alternatively, service providers may create subclasses of the OWL-S classes and instantiate those instead).

Service, with some basic concepts that tie the parts of an OWL-S service description together and holds a textual description of the service.

Profile, which describes what it provides to clients, and what it requires of them. More specifically, a service profile presents the inputs, outputs, preconditions and effects of a service. This information is used for matchmaking, i.e. to find an appropriate service based on its capabilities.

Process, which has properties used to describe how the service works, i.e. what happens when the service is used. Services can be described as a collection of atomic or composite processes, which can be connected together in various ways, and the data and control flow can be specified.

Grounding, with properties to specify how the service is activated, including details on communication protocols, message formats, port numbers, etc. This *abstract grounding* is usually tied to a *concrete grounding* in the form of a WSDL interface description.

The Web Service Modeling Ontology (WSMO) [Rom05] allows specification of extra-functional properties for each particular element of a Web service description. It covers a bigger list of such properties including; accuracy, contributor, coverage, creator, date, description, financial, format, identifier, language, network-related QoS, owner, performance, publisher, relation, reliability, rights, robustness, scalability, security, source, subject, title, transactional, trust, type, version. However, analysis on these properties is not possible because the properties are not included in the logical model of WSML. The Web Service Modeling Language (WSML) is a language for WSMO.

Electronic Business (eb-*)

The ebXML (electronic business XML) [WD06] is a framework that provides a global electronic market place where enterprises of any size, anywhere can find each other electronically and conduct business through exchange of XML based business messages. It is a standardisation effort established by the United Nations body for Trade Facilitation and Electronic Business (UN/CEFACT) and the Organisation for the Advancement of Structured Information Standards (OASIS). ebXML consists of several technologies which are provided in five main modules in the architecture. Some of these modules can be used individually and they define several aspects of service contracts.

Business Process Specification Schema (BPSS) describes collaboration between business partners, their roles, relationships and responsibilities. It defines the choreography of business documents thus covering the same domain abstractly as BPEL. The roles (of business partners) interact with each other through Business Transactions. The business transactions form a Choreography and each Business Transaction consists of one or two document flows. An example ebXML BPSS specification of a simple notification transaction with one document flow is given below. The example is adapted from the Business Process Specification Schema document [CCK+01].

```
<BusinessTransaction name="Notify of advanceshipment">
    <RequestingBusinessActivity name="">
       <DocumentEnvelope
           BusinessDocument name="ASN"/>
    </RequestingBusinessActivity>
    <RespondingBusinessActivity name=""
    </RespondingBusinessActivity>
</BusinessTransaction>
```

Consequently, ebXML BPSS defines two kinds of collaborations from the defined business transactions, binary and multi-party collaboration. Before

we illustrate further, let us first introduce how the contracts between two parties are defined (CPA) and how the capabilities of a company are described (CPP).

Collaboration Protocol Profile (CPP) constrains the interaction of partners by describing the capabilities of an individual party through *Business capabilities* which describe business processes and *Technology capabilities* which describe message exchange capabilities, transport and security constraints.

Collaboration Protocol Agreement (CPA) expresses an agreement between partners. Usually, CPA is derived from CPPs of trading partners. It describes the capabilities that trading partners have agreed to use to perform a particular business collaboration. In other words, it is a contract between two or more trading partners. CPA is also used by trading parties' computing systems to set up a runtime environment for the exchange of business messages. Security characteristics of business process collaboration are also defined in BPSS, CPP and CPA.

As mentioned above, BPSS specifies a binary and a multi-party collaboration. A Binary Collaboration is always between two roles. These two roles are called Authorized Roles, because they represent the actors that are authorized to participate in the collaboration. The CPA/CPP Specification requires that parties agree upon a Collaboration Protocol Agreement (CPA) in order to transact business. A CPA associates itself with a specific Binary Collaboration.

A multi-party collaboration is a synthesis of binary collaborations. A multi-party collaboration consists of a number of business partner roles. Each binary pair of trading partners will be subject to one or more distinct CPAs.

```
<MultiPartyCollaboration name="DropShip">
  <BusinessPartnerRole name="Customer">
   <Performs initiatingRole=
     //binaryCollaboration[@name="Firm Order]
   /InitiatingRole[@name=buyer]/>
  </BusinessPartnerRole>
  <BusinessPartnerRole name="Retailer">
    <Performs respondingRole=
      //binaryCollaboration[@name="Firm Order]
     /RespondingRole[@name=seller]/>
    <Performs initiatingRole=
      //binaryCollaboration[@name=" Product
   Fulfillment /InitiatingRole[@name=buyer]/>
  </BusinessPartnerRole>
  <BusinessPartnerRole name="DropShip Vendor">
    <Performs respondingRole=
      //binaryCollaboration[@name=" Product
         Fulfillment
       /RespondingRole[@name=seller]/>
    </BusinessPartnerRole>
</MultiPartyCollaboration>
```

Table 1: Classification of Service Contract Specification Languages.

Aspects		Contracts Languages/Approaches		
		Web Services (WS-*)	Semantic Web (*-S)	Electronic Business (eb-*)
Interface		WSDL	OWL-S	ebBSI
Functionality		WS-BPEL, WSOL	OWL-S (IOPE), WSMO	ebBPSS
Protocol		WS-BPEL, WS-CDL	WSMO	ebBPSS
Security		WS-Security		ebCPA(SecurityPolicy)
Quality	policy	WS-Policy		
	trust	WS-Trust		ebCPP(XMLDSIG)
	availability	WSOL		
	performance	WSLA, WSOL	WSMO/WSML	ebCPA

Each Business Partner Role performs one Authorized Role in one of the binary collaborations, or perhaps one Authorized Role in each of several binary collaborations. This is modeled by use of the Performs element. This *Performs* linkage between a Business Partner Role and an Authorized Role is the synthesis of Binary Collaborations into Multiparty Collaborations. Implicitly the Multiparty Collaboration consists of all the Binary Collaborations in which its Business Partner Roles play Authorized Roles [CCK+01].

Summary: Having presented service contract languages from the three families dealing with service based application, we now present in the next section a classification of these languages.

5 Classification of Contract Languages

Several notations and languages that describe service contracts are presented in the previous section. Each of the notations cover one or more aspects of service contracts such as interfaces, functionality, behaviour, security, and quality. Here, we propose a classification of the notations/languages based on the aspects they cover. It is summarized in Table 1.

Each of the three families uses a set of notations/languages to specify different aspects of service contracts. The Web Services family has the WSDL at its core with many other languages and extensions more than the other two families. The Semantic Web family employs two major ontology based languages, OWL-S and WSMO thus covering many aspects in one language unlike the many extensions of the WS-* family. In relation to the WS-* family, OWL-S atomic processes correspond to WSDL *operation*s and each of the set of inputs and outputs of an OWL-S atomic process correspond to WSDL *message*s. The Electronic Business family employs a number of XML schema to capture various aspects of service contracts under a common ebXML framework. In relation to the WS-* family, BPSS covers the same domain as BPEL.

All the three families agree that the aspects are needed in addition to having an XML grammar. Each of the families seems to be viable as it employs a set of notations to cover different aspects. The Web Services and Electronic business family share an operational programming style whereas the Semantic Web family has a predicative style. One may view OWL-S as a specification language (functionality) for the others. However, working with predicate logic is not attractive in protocols, security and performance. This implies that there is a gap between the Semantic Web family and the others. The WS-* family and eb-* family are modeling languages as opposed to property languages.

Analysis Models for the Classes

Interface Aspect: These can be analyzed by standard type checking as done by XML parsers. It is a form of matching by identity. However, there is no concept of superclass or subclass. On the whole, the three families are based on XML and therefore share this analysis technique.

Functionality Aspect: State machines (including Petri nets and ASML) can be used to model functionality in WS-* and eb-* families and properties can then be analyzed by model checking or simulation. The *-S family which is based on predicates require an implementation in some programming language which then can be analyzed by program verification (cf. LOOP project [JP04] and JML/Java [LBR06]).

Protocol Aspect: Process algebra which are usually modeled by means of labeled transition systems; concrete examples are CSP and CCS and Petri nets apply here. In the Semantic web family, the logic based models of OWL-S is a bit unclear with predicative and concurrency aspect. A predicative style of analysis such as UTP [HH98] style using trace logic may apply.

Security Aspect: The security aspect as captured in WS-* family and eb-* security tokens is prescriptive, thus might not be aimed at analysis but on monitoring.

Quality Aspect: The quality aspect as it stands is conceptual and similar to the security aspect and may be aimed more at monitoring.

In summary, the WS-* and eb-* families agree to an extent such that it seems practical to build bridges between services that span them. The *-S family is taking another direction which may indicate that it is perceived as a higher level predicative notation. It may be able to specify essential commonalities of contracts from other families; although that remains to be seen.

6 Conclusions

A classification of service contract languages including some of the aspects of service contracts and related analysis models currently being explored in service-based application domain is presented. Each of the three families in the classification tries to cover the different aspects of service contracts. This implies that they are viable. In our study, we did not find anything technical that prevents the merging of the Web Services and Electronic Business families. The Semantic Web family is based on a predicative style and is not operational. It is not clear how the predicative style will handle protocol aspects. However, it is possible to have a one way mapping of some aspects, for instance a mapping from a functional aspect specified in BPEL to OWL-S processes. In this case model checking may apply.

Where as there is the possibility of merging the WS-* and eb-* family, there is a gap in the *-S family. The Semantic Web family is a property language and can be used to specify for example, functionality for the others. Further, it is based on predicate logic which is not attractive in protocols, security and performance.

As a basis for further work on detailed analysis, we are adopting the WS.* family, because the eb-* family targets a smaller community. We look forward to seeing how the Semantic Web family develops as property language.

Acknowledgment

We thank the reviewers for helpful comments.

Paper B: Consistency Checking of Web Service Contracts[2]

Authors:
M. Emilia Cambronero
University of Castilla-La Mancha, Spain

Joseph C. Okika and Anders P. Ravn
Aalborg University, Denmark

Abstract

Behavioural properties are analyzed for web service contracts formulated in Business Process Execution Language (BPEL) and Choreography Description Language (CDL). The key result reported is an automated technique to check consistency between protocol aspects of the contracts. The contracts are abstracted to (timed) automata and from there a simulation is set up, which is checked using automated tools for analyzing networks of finite state processes. Here we use the Concurrency Work Bench. The proposed techniques are illustrated with a case study that include otherwise difficult to analyze fault handlers.
Keywords: Web Services contract, consistency, WS Choreography, WS Orchestration.

[2]This chapter is previously published in [COR08].

1 Introduction

Service Oriented Architecture (SOA) [Erl05] reorganizes series of previously operational software applications and support infrastructure into an interconnected set of services, each accessible through standard interfaces and messaging protocols. It promotes services that are distributed, heterogeneous, autonomous and open in nature. SOA is particularly applicable when multiple applications running on varied technologies and platforms need to communicate with each other. With SOA, enterprises can mix and match services to perform business transactions with less programming effort. SOA is implemented with web service technology. Thus there is consensus today, that a web service is a programmable component that provides a service and is accessible over the Internet. They are based on standards like Simple Object Access Protocol (SOAP) [LM07a, See01, KGH+07], can be standalone, or linked together to provide enhanced functionality.

Businesses depend on web services, therefore their properties are of great importance, and informal checking and consensus approaches to when a service is good enough may not suffice. A business will only reluctantly use enterprise applications offered as open web services, because of the high risks involved in using untrusted services from unknown providers. Formal contracts defining the desired properties are therefore studied intensively today, because they are a way to manage the risks that come with the interaction among these inter-organizational services.

Traditionally, contracts in an object oriented setting consider only the functional aspect (pre-condition, post-condition, invariant) of an interface specification. A pre-condition is a constraint that must be satisfied before calling a method or operation; it checks for valid arguments. A post-condition is a corresponding property that is true when the call completes; it is the input-output relation. Finally, an invariant is a constraint on the state of an object; it must hold before and after any operation, and clearly after initialization of the object. These concepts, as popularized by Meyer's "Design by Contract" [Mey97], are, however, just part of the properties exhibited by web services. Since web services are intrinsically distributed, they are by nature concurrent programs, and thus their overall functionality depends not only on correct implementation of the local functionality by sequential algorithms, but even more on the interplay between local functionality and global behavior (protocols and timing).

In this paper we focus on protocol or behavioural aspects of service contracts. There are several proposals for contract specification standards for web services, see e.g. [OR08] for an overview. Prominent among these standards are the Business Process Execution Language (WS-BPEL)[ACD+03]

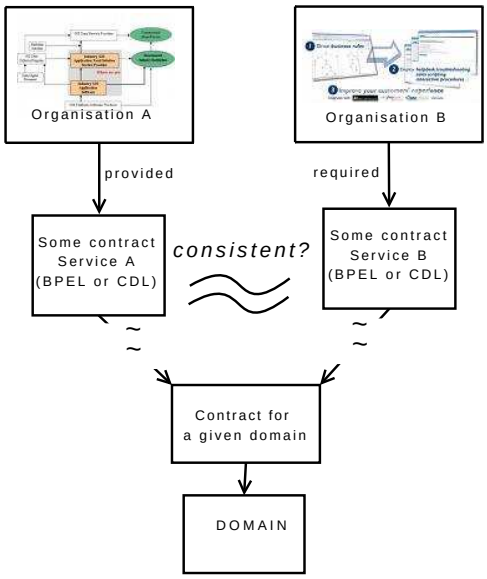


Figure 1: Analysis of Web Service Contracts

and Choreography Description Language (WS-CDL) [KBR+04]. BPEL offers a programming model for specifying the orchestration of web services whereas CDL specifies the choreography of interacting services. However, when web service contract are specified using either BPEL or CDL, there is no assurance that they are consistent unless verified. Though there are efforts toward this form of analysis, there remain challenges in the area of automated approach to checking consistency in addition to other properties.

In previous work [DPC+05] we have demonstrated a viable solution to the problem of checking for functional and behavioural properties of individual services. This is done through translation of the specifications to timed automata followed by model checking for relevant properties. In [COR07] we considered the problem of consistency across specifications and identified a need to set up a correspondence between the individual automata. The novel contribution in this paper is to make such a consistency check practical by translating the automata to CCS, the input language for the Concurrency Work Bench. As demonstrated by a case study, this technique is applicable and gives a handle for automating yet another consistency check for web services.

Directly Related Work

Web Service contracts is attracting a lot of attention and several researchers propose various approaches and frameworks toward specification and anal-

ysis. For instance [CGP07, CCLP06, DKR04, PZWQ06] looks at it from a formal semantics viewpoint, whereas [RGWC99, PS07] propose languages for specifying contracts. All these points to the fact that there is an important need for contracts to be specified and analyzed.

An earlier treatment of contracts in an object-oriented paradigm is Design by Contract [Mey97]. Similar treatment concerning components is found in [BJP99]. Here, the functional specification is achieved through assertions; which consists of preconditions, post-conditions and invariants. The framework in [Mey92] takes a pragmatic approach at code level where the assertions are part of the language. We agree that these functional specifications are important in order to specify a formal agreement between a service provider and its clients. It expresses what a client should do before making a service request and what the provider will give as result of it.

Among the related work of Web Service contracts is [HL05]. It proposes to visualize contracts by graph transformation rules. Apart from expressing contracts in terms of pre- and post-conditions of operations together with invariants, they introduced the notions of provided and required contracts. With this, they use the provided contracts to create the test cases and test oracles whereas the required interfaces are used to drive the simulation. We like their treatment of functional specifications, but it needs to be supplemented with other aspects, and one may gain something by investigating model checking as a supplement to testing.

Quantitative aspect are researched in [KL03, WS-04, wsa04]. The Web Service Level Agreement (WSLA) framework [KL03] is targeted at defining and monitoring SLAs for Web Services. WSLA enables service customers and providers to unambiguously define the agreed performance characteristics and the way to evaluate and measure them. We want to mention here that WSLA complements Web Service Definition Language (WSDL) [CCMW01, BL06], which is an XML grammar that describes the capabilities of Web services through its interface descriptions. WSLA is used to define a contract between service provider and service requester, but its treatment of functional behavior is limited.

The above mentioned contributions focus on a single web service language, and either the functional or the behavioral side of a contract. We extend their perspective by considering the overall consistency of a service specified in languages covering more than one aspect. Furthermore we demonstrate how existing tools are adapted for such checks.

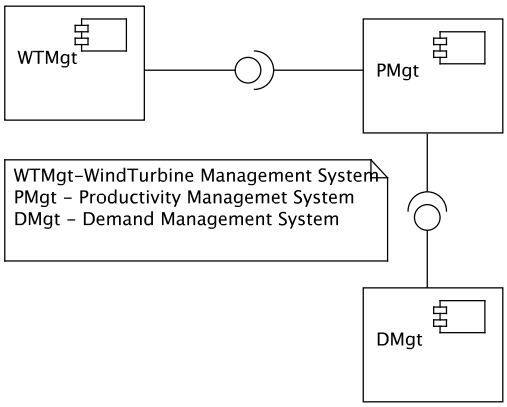


Figure 2: Wind Turbine Management System Components

Overview

In Section II, we give a detailed presentation of Web Service contracts where the aspects of contracts are described. We introduce in this section, a case study of a Windmill Management System. Section III details the analysis of Web Service contracts. General consistency, satisfiability, and application specific issues are presented. A comparison with other approaches follows and finally, we conclude in Section V.

2 Web Service Contracts

To manage the risks that come with the interaction among several services, the service provider and a consumer must have a contract that specifies the details of the service. As mentioned before, it is important to note, however, that there are different aspects of contract in play when dealing with web services. First, there is the functional aspect which describes the functional properties, and second, there is the protocols aspect which specifies the behaviour as a sequence of messages, events, signals, etc. There is also the extra functional QoS (Quality of Service) requirements aspect. This is further illustrated following the example presented in the following subsection.

Example

We consider a Windmill Management System. The system monitors and controls wind turbines, and it has several components which are web services located in different places. We focus on three of these components, because

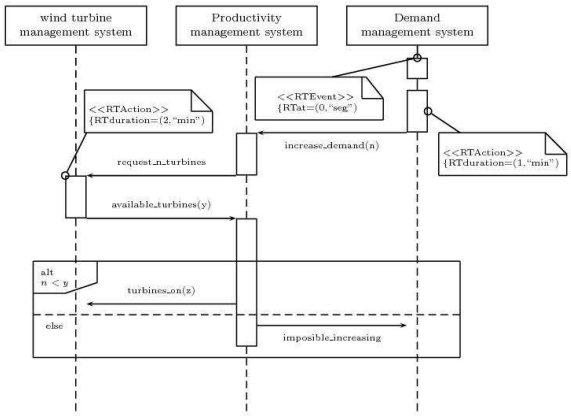


Figure 3: Wind Turbine Management System Sequence Diagram

it gives us the scenario needed to specify a web service contract. The components are briefly described below and shown as an UML component diagram in Figure 2. The interaction between these services are illustrated using a RT-UML sequence diagram, shown in Figure 3. The informal requirements for the components are:

- Wind Turbine Management: sends a report to Productivity management every hour.
- Productivity Management: receives and analyzes the report from Wind Turbine Management.
- Demand Management: generates a report of power needs for Productivity Management.

We look at this example from two perspectives; WS-CDL and WS-BPEL. WS-CDL provides a definition of the information formats being exchanged by all participants. In other words, it specifies the protocols. WS-BPEL provides the message exchanges and functions as viewed by one participant. It describes the functionality of a single business process offered as a service by an enterprise.

Contract Aspects in WS-CDL

CDL offers a model for specifying a common understanding of message exchanges. This language describes the choreography of web services systems, that is, the relationships between the composite services in a peer-to-peer environment. It uses the WS definition language (WSDL) to define and locate common type definitions.

WS-CDL is a very verbose notation, therefor the key concepts of contracts in WS-CDL are summarized below, while a full description of the demand management system is found in appendix A.

Interface

In WS-CDL, each interface is associated with a particular role, where a `roleType` enumerates potential observable behaviors a participant can exhibit when interacting with other participants. The syntax is the following:

```
<roleType name="DemandRoleType">
      <description type="description"/>
      <behaviour name="DemandBehaviour"
           interface="WSDLDemandType" />
</roleType>
```

The `behaviour` element defines an optional interface attribute, which identifies a WSDL interface type.

Functional Specification: pre-conditions, post-conditions and invariants

In WS-CDL these elements are defined by means of *workunits*; which define the constraints that must be fulfilled for making progress and describe some activities within a choreography. The constraints are give by *XPath 2.0* expressions.

XPath 2.0 supports date and time variables, so we can use these variables in WS-CDL as well. Furthermore, XPath provides a number of functions to manage these datatype values.

```
<workunit name="demand increase detected"
      guard="cdl:equal(cdl:getVariable
         ('tns:DemandClock'),'','),'0:00')"
      block="true">
  <assign roleType="DemandRoleType">
    <copy name="calculateincrease"
          causeException="true">
      <source variable="true"/>
      <target variable=
       "cdl:getVariable('detectedincreaseDone','','')"/>
    </copy>
  </assign>
</workunit>
```

A *workunit*'s `guard` element establishes the condition, which has to be fulfilled to perform the workunit activities. This element allows us to define pre-conditions. Postconditions and invariants can be introduced by appending a workunit with the condition as a guard at the end of the normal workunit flow. In order to define a condition we use XPath and XML Schema expressions.

Protocol

A *sequence* of activities is modeled in WS-CDL using the ordering structure `sequence`, which contains a set of activities that can perform sequentially.

A *non-deterministic choice* is implemented in WS-CDL using the ordering structure `choice`. The WS-CDL standard says that when two or more activities are specified here, only one of these is selected and the other ones are disabled. It is assumed that the selection criteria for those activities are non-observable.

The following WS-CDL code corresponds to the fragment in which the productivity system sends a message to the turbine system for the turbines to be turned on or else it sends a message to the demand system to indicate that it is not possible to satisfy the new demand. As you can see, it is modeled in WS-CDL by a choice activity in which we have two activities, and only one of them can be finally executed.

```
<choice>
  <workunit name="alt_else1_if"
            guard="Available == true" block="true">

      <interaction name="TurbinesOn_interaction"
            operation="TurbinesOn"
            channelVariable=
                 "Productivity2WindTurbineChannel">
        <participate relationshipType=
                    "ProductivityWindTurbine"
             fromRole="ProductivityRoleType"
             toRole="WindTurbineRoleType"/>
        <exchange name="TurbinesOnExchange"
                  action="request"/>
      </interaction>
  </workunit>

  <workunit name="alt_else1_else"
           guard="Available != true" block="true">

      <interaction name="Imposible_interaction"
                operation="Imposible"
                channelVariable=
                       "Demand2ProductivityChannel">
           <participate relationshipType=
                           "ProductivityDemand"
                fromRole="ProductivityRoleType"
                toRole="DemandRoleType"/>
           <exchange name="ImposibleExchange"
                     action="request"/>
      </interaction>
  </workunit>
</choice>
```

An *external choice* is implemented in WS-CDL using the ordering structure *workunit*, since it allows us to establish conditions to execute the corresponding activity. For that purpose, we may use the guards of workunits, by including in a guard an expression related with the value of a variable.

In WS-CDL, we use the workunit `repeat` to implement repetition. A workunit that completes successfully must be considered again for matching (based on its guard condition), if its repetition condition evaluates to `true`.

```
<workunit name="alt_else1_if"
         guard="Available == true"
         repeat="false" block="true" >

   <interaction name="TurbinesOn_interaction"
         operation="TurbinesOn"
         channelVariable=
             "Productivity2WindTurbineChannel">
       <participate relationshipType=
                       "ProductivityWindTurbine"
          fromRole="ProductivityRoleType"
          toRole="WindTurbineRoleType"/>
       <exchange name="TurbinesOnExchange"
                    action="request"/>
   </interaction>
</workunit>
```

Timing

Lower bounds, upper bounds, explicit clocks, reset and stop operations are handled by XPath and XML Schema.

XPath 2.0 supports date and time variables, so we can also use these variables in WS-CDL. Actually, XPath provides a number of functions to manage these datatype values. These variables can be used in particular to delay the execution for a certain time, or to establish the instant at which some actions must be executed. For that purpose, we may use the guards of workunits, by including in a guard an expression related with the value of a time variable.

Specifically, we use the XPath and XML Schema notation to specify the time aspects as follows:

Explicit clocks are introduced by `xs:time`.

Bounds are specified inside a workunit guard. In fact, as we capture delays or instants of execution, the specific expressions allowed are those constructed using the operators `op:time-equal` `op:time-less-than` and `op:time-greater-than` of XPath 2.0. We can also use the `hasDeadlinePassed` operation, which is defined in the WS-CDL specification to manage timing.

Reset. In WS-CDL we reset a clock using an `assign` activity, which creates or changes the variable defined by the target element using the expression defined by the source element (in the same role).

Stop. In order to model that a clock is stopped, we can capture the value of the time, of this specific instant, in a clock variable and then, when we want to initiate the time again, we can use the clock variable to continue from this point. We use two `assign` activities to capture and change the time value.

Synchronization. The `interaction` WS-CDL element defines how the parties in a web services are synchronized. An interaction activity involves two roletypes, and an exchange of information between them. Actually, in WS-CDL several exchanges of information are allowed in a single interaction, and they can be either `request` or `respond` types, and these actions can be synchronous or asynchronous, depending on the `align` attribute.

```
<interation name="The demand management system
           sends increase in power demand to
           the productivity system"
   operation= = "sendIncreasing"
   channelVariable="Demand2ProductivityC">
   <description type="description">
     Sending the necessary increase of demand
   </description>
   <participate
      relationshipType= "DemandProductivity"
      fromRole="DemandRoleType"
      toRole="ProductivityRoleType" />
   <exchange name= "CalculatedIncerasing"
     informationType="Increase_demandType"
     action="request">
   </exchange>
   <timeout
     time-to-complete= "cdl:minor(cdl:getVariable
       ('tns:Clock1','',''),'1:00')">?
</interaction>
```

In the `time-to-complete` attribute the timeframe in which an interaction must complete is specified. Then, when this time expires (after the interaction was initiated) and the interaction has not completed, a timeout occurs and the interaction finishes abnormally, causing an exception block to be executed in the choreography. The optional attributes `fromRoleTypeRecordRef` and `toRoleTypeRecordRef` are XML-Schema lists of references to record elements that will take effect at both roleTypes of the interaction.

Faults

Choreographies may have one exception block, which consists of some (possibly guarded) *workunits*, but only one of them can be finally executed (the first one whose guard evaluates to true). When the exception block is executed, the choreography terminates abnormally, even if the default exception workunit has terminated correctly. Exceptions are the following:

Interaction failures For instance, sending of a message failed.

Timeout errors For instance, an interaction did not complete within the alloted time.

Application failures These are for instance illegal expressions.

CDL in summary

Overall CDL is a coordination language which focuses on the communication between agents providing the services. It is therefore very appropriate to give it a semantics by translation into a network of communicating processes.

Contract Aspects in WS-BPEL

BPEL is a programming language to specify the behavior of a participant in a choreography. It allows existing Web services to be orchestrated into composite services. Choreography is concerned with describing the message interchanges between participants.

WS-BPEL is verbose also, so we do not include full descriptions; but as for WS-CDL, we present the WS-BPEL contract aspects below:

Interface

In WS-BPEL, the services with which a business process interacts are modeled as `partnerLinks`. Each `partnerLink` is characterized by a `partnerLinkType`, which defines the roles played by each of the services in the conversation and specifies the `portType` provided by each service to receive messages within the context of the conversation. These `portTypes` are defined in the WSDL document, and each role specifies exactly one WSDL `portType`.

In order to utilize operations via a `partnerLink`, the binding and communication data, including *endpoint references (EPR)*, for the `partnerLink` must be available. The fundamental use of endpoint references is to serve as the mechanism for dynamic communication of port-specific data for services. An example fragment of a `partnerLink` is:

```
<partnerLinks>
<partnerLink name="productivity">
partnerLinkType="as:productivityDemandMSLT"
   myRole="DemandMS"
   partnerRole="productivity" />
</partnerLinks>
```

The endpoint references syntax is:

```
<service-ref reference-scheme="http://example.org">
  <foo:barEPR xmlns:foo="http://example.org">
   ... </foo:barEPR>
</service-ref>
```

Functional Specification: preconditions, postconditions and invariants

WS-BPEL uses several types of expressions to implement the functional part of a web service contract:

- Boolean expressions. These expressions can appear inside a transition, a join, a while, and an if condition.
- Deadline expressions. The WS-BPEL elements that use these expressions are until-expressions of onAlarm and wait.
- Duration expressions. These appear in the `for` expression of `onAlarm` and `wait`, and the `repeatEvery` expression of `onAlarm`.
- Unsigned Integer expressions, that include counter values `startCounterValue`, `finalCounterValue`; as well as branches in a `forEach`.
- General expressions inside assign activities.

Protocol: sequence, choice, and iteration

- A sequence of activities is modeled by the `sequence` structured activity. It contains one or more activities that are performed sequentially, in the lexical order in which they appear.

  An example is the Productivity process which is given as a sequence as follows:

```
<sequence>
 <if
 bpel:getVariableProperty('x','time:level')==0>
   <then>
  <!-Process productivity (invoke) - ->
   <assign>
<copy>
<from partnerLink="productivityMS"
endpointReference="myRole" />
<to>&increaseData.productivityMSRef </to>
</copy>
 </assign>
```

```
 <invoke name="increaseDemand"
          partnerLink="productivity"
          portType="as:productivityPT"
          operation="process"
          inputVariable="increaseData">
   <correlations>
   <correlation set="increaseIdentification"
   </correlations>
 </invoke>
 </if>
</sequence>
```

- Choice. Both non-deterministic and external choice are expressed in WS-BPEL by means of `pick` activities, which waits for the occurrence of an event and then executes the activity associated with it. When several events occur simultaneously, an implementation dependent choice is made. Thus, in analysis, the choice must be modeled as non-deterministic.

- Conditional. WS-BPEL contains a conventional conditional statement as well.

- Iteration. WS-BPEL uses the `while` and `repeatUntil` activities, to model iteration.

```
<while>
   <condition>
      $numberWindTurbine < 10
   </condition>
   <scope>  ... </scope>
</while>

<repeatUntil standard-attributes>
    standard-elements
    activity
    <condition expressionLanguage="anyURI"?>
       ... bool-expr ...
    </condition>
</repeatUntil>
```

Timing

Lower bounds, upper bounds, explicit clocks, reset and stop operations are specified as in WS-BPEL using XPath and XML Schema.

Explicit clocks, lower and upper bounds They are defined using XML Scheme notations, as explained before.

Reset In WS-BPEL we can reset the clock using an `assign` activity, which copies data from one variable to another.

```
<assign validate="yes|no"? standard-attributes>
   standard-elements
     (<copy keepSrcElementName="yes|no"?>
       from-spec  to-spec  </copy> |
      <extensibleAssign>
        ...assign-element-of-other-namespace...
      </extensibleAssign>) +
</assign>
```

Stop In order to model that a clock is stopped in WS-BPEL we do as in WS-CDL.

Concurrency and Synchronizations They are implemented in WS-BPEL using a `flow` activity, which provides concurrency and synchronization. A `flow` completes when all of the activities enclosed by it have completed.

```
<flow standard-attributes>
    standard-elements
    <links>? <link name="NCName"> </links>
    activity+
</flow>
```

Faults

Business processes are usually of long duration. They can manipulate data in back-end databases and line-of-business applications. Error handling in this environment is both difficult and business critical. The overall business transaction can fail or be canceled after many transactions have been committed. In this cases, the partial work done must be undone or repaired as best as possible. Error handling in WS-BPEL processes therefore leverages the concept of compensation, that is, application-specific activities that attempt to reverse the effects of a previous activity that was carried out as part of a larger unit of work that is being abandoned. It thus provides the means for a forward error recovery.

Specifically, WS-BPEL provides constructs to declare fault handling and compensation.

Compensation handler WS-BPEL allows scopes to delineate that part of the behavior that is meant to be reversible in an application-defined way by specifying a compensation handler. A `compensationHandler` is simply a wrapper for an activity that performs compensation.

```
<compensationHandler>
   activity
</compensationHandler>
```

It is invoked with `compensateScope`, when an explicit scope is compensated, or `compensate` when successfully completed inner scopes are compensated in reverse order. A compensation handler for a scope is available for invocation only when the scope completes successfully.

```
<compensateScope target="NCName"  standard-attributes>
   standard-elements
</compensateScope>

<compensate standard-attributes>
   standard-elements
</compensate>
```

Compensations may only be invoked in `catch`, `catchAll`, `compensationHandler` and `terminationHandler` activities, where termination handlers provide the ability for scopes to control the semantics of forced termination by disabling the scope's event handlers and terminating its primary activity and all running event handler instances.

Fault handling In a business process it can be thought of as a mode switch from the normal processing in a scope. Fault handling in WS-BPEL is designed to implement backward error-recovery in that it aims to undo or repair the partial and unsuccessful work of a scope in which a fault has occurred. The completion of the activity of a fault handler, even when it does not rethrow the handled fault, is not considered successful completion of the attached scope. Compensation is not enabled for a scope that has had an associated fault handler invoked.

Explicit fault handlers attached to a scope provide a way to define a set of custom fault-handling activities, defined by `catch` and `catchAll` constructs. Each `catch` construct is defined to intercept a specific kind of fault, defined by a fault `QName`. If the fault name is missing, then the catch will intercept all faults with the same type of fault data. A `catchAll` clause can be added to catch any fault not caught by a more specific fault handler.

```
<faultHandlers>
   <catch faultName="QName"?
      faultVariable="BPELVariableName"?
      ( faultMessageType="QName" | faultElement="QName" )?>*
      activity
   </catch>

   <catchAll>?  activity  </catchAll>
</faultHandlers>
```

There are various sources of faults in WS-BPEL. A fault response to an `invoke` activity is one source of faults, where the fault name and data are based on the definition of the fault in the WSDL operation. A `throw` activity is another source, with explicitly given name and/or data. WS-BPEL defines several standard faults with their names, and there may be other platform-specific faults such as communication failures.

BPEL summary

BPEL is essentially a programming language. However it has some features that are specially tailored to make it easier to build robust systems that can recover from a variety of faults. It includes features for expressing internal concurrent activities; they should however be used with care, because it is not always easy to comprenhed the interaction with compensations and fault handlers.

3 Analyzing Web Service Contract

Having described all the elements of specifications, we now present the translation to automata. In order to perform this translation, we note that WS-CDL and WS-BPEL are XML based languages for describing Web Services. The timed automata formalism we use is UppAal [BLL+96]; and it is represented by another XML document, thus, the translation has been developed with XSLT [Cla98], XML Style sheets Language for Transformation, which is a language for transforming XML documents into other XML documents.

Figure 4 shows how the translation works: we have created some XSL style sheets, where we use XSLT instructions to extract the information from the WS-CDL document, and then the UppAal document is automatically generated. This document can be opened with the UppAal tool, and thus, we can use the model-checker of UppAal to verify some properties of interest. The tool can also run simulations of the model. We have also created some XSL style sheets to perform the same translation for WS-BPEL documents.

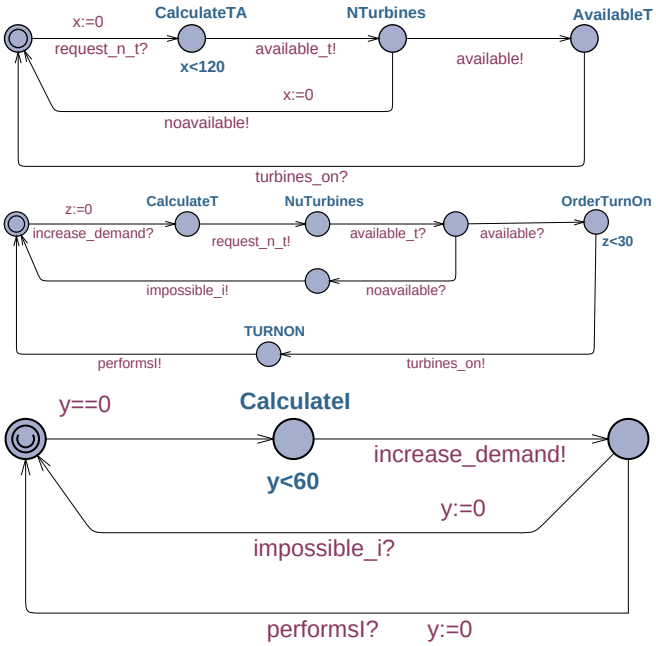
For the two aspects we can check the following.

General Properties

We check the absence of deadlock for the CDL and for the BPEL; thus we check that the system is able to progress from start to termination; in UppAal this is easily formulated:

$A[]$*not deadlock*

This property holds for both systems.



A. WindTurbineMS

B. ProductivityMS

C. DemandMS

Figure 4: Wind Mill Management System modeled in UppAal

The system should also be useful. If there are enough available turbines to fulfill the increase of demand, then the Productivity Management system shall send the command to turn on some of them to the Wind Turbine management system. This is formulated as the invariant that says that for all computations (A) and for all states ([]), the two automata locations coincide:

$A[]\ WindTurbineMS.AvailableT \rightarrow$
$ProductivityMS.OrderTurnOn$

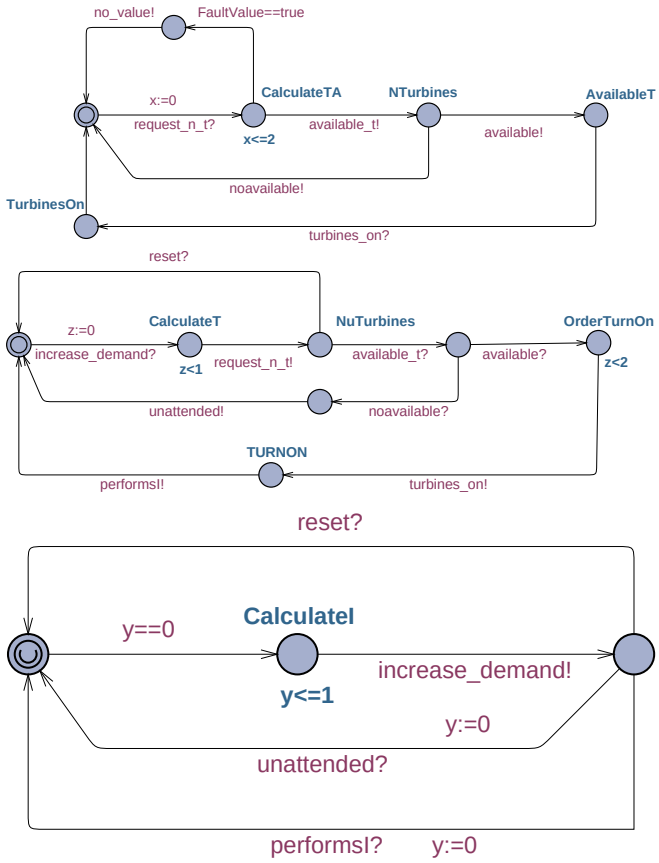
This example property holds as well.

Meeting the demand

Here we check for a BPEL property that the methods can be executed satisfying the contracts or generating the exceptions. For instance, when the demand system sends a message to the productivity system, because it detects an increase in the power demand (the message *increase_demand*). Also, the Wind Turbine Management system always sends the number of available turbines on Productivity Management system's demand. This is represented in UppAal as follows:

$A[]\ ProductivityMS.NuTurbines \rightarrow$
$WindTurbineMS.CalculateTA$

which holds as well.



A. WindTurbineMS-BPEL

B. ProductivityMS-BPEL

C. DemandMS-BPEL

Figure 5: Wind Mill Management System modeled in UppAal - from BPEL

Model checking summary

The form of checking that has been shown above is really exhaustive testing. Analysis of what properties to check depends on a systematic inspection of both requirements and the design by some review process, for instance Software Reviews, Code Inspections, and other proactive management processes whose purpose is to eliminate or to find and remove errors in product design as early as possible.

4 Consistency Checking - Simulation

To check whether the two individually derived models are consistent, we use the concept of (bi-)simulation. A (bi-)simulation is an equivalence relation between state transition systems, associating systems which behave in the same way in the sense that one system simulates the other and vice-versa.

The automata generated from the two contract aspects specification systems (WS-CDL, WS-BPEL) turn out to be bi-similar in the following aspects:

- they both accept the same operation sequence; since the WS-CDL specified the protocols, while WS-BPEL contains the operation names but with more information.
- they also accept the same message sequence. Thus, the state that receives the message (e.g. *increase_demand* in the example in Figure 4) is followed by a state that sends the message (*request_n_t*) in both automata. The automaton from WS-BPEL may contain some internal states.

We use another model checking tool CWB-NC to check the consistency. We first map the contract captured by both BPEL and CDL to CCS [Mil89], one of the the design languages for CWB, which has communication similar to UppAal; actually, UppAal was developed by people who had prior experience with CCS and the Concurrency Workbench. With the analogous roots, we have not found it useful to spend much time on whether this simple mapping preserves the semantics; it is fairly obvious that it does. More languages such as timed actions version of CCS, CSP, basic LOTOS, etc are supported as well in the CWB tool; it performs model checking, preorder checking and equivalence checking. As mentioned above, we focus on equivalence checking which allows to identify the behaviourally/observationally equivalent states in a system.

One may ask, why CWB is not used throughout the analysis, since it includes model checking. The answer lies in the lack of state variables; CWB can model the communication structure only, whereas UppAal supports state variables with bounded domains as well as clocks.

Translation from Uppaal to CWB CCS (CDL)

We translate the contract specification models in UppAal to a process algebra CCS to allow us to check consistency. The Wind Mill management system consists of 3 processes as shown below:

```
proc WTMCDL = (WMC | DMC | PMC)\
{request_n_t, available_t,
 noavailable, available,
 increase_demand, unattended,
 performsI}
```

Processes WMC, DMC, and PMC correspond to Windturbine management system, demand management system and productivity management

system respectively as modeled in Figure 5. The three processes communicate through synchronization events. For instance, *request_n_t* in Windturbine management and productivity management.

Translation from Uppaal to CWB CCS (BPEL)

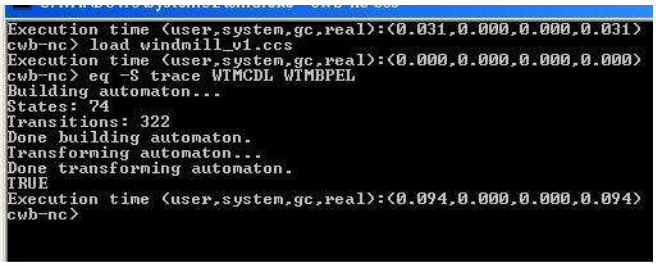
Similar to the translation of CDL, we translate the contract specification models in UppAal to a process algebra CCS. However, we have more processes from the BPEL contract specifications. These additional processes are fault handlers, compensation handlers and event handlers; but we focus on a fault handler. One can easily add other processes without violating consistency, since they are abstracted away when checking against CDL. In this case, the Wind Mill management system consists of 4 processes as shown below:

```
proc WTMBPEL = (WMC | DMC | PMC | FH)\
{fault, reset
 request_n_t, available_t,
 noavailable, available,
 increase_demand, unattended, performsI}
```

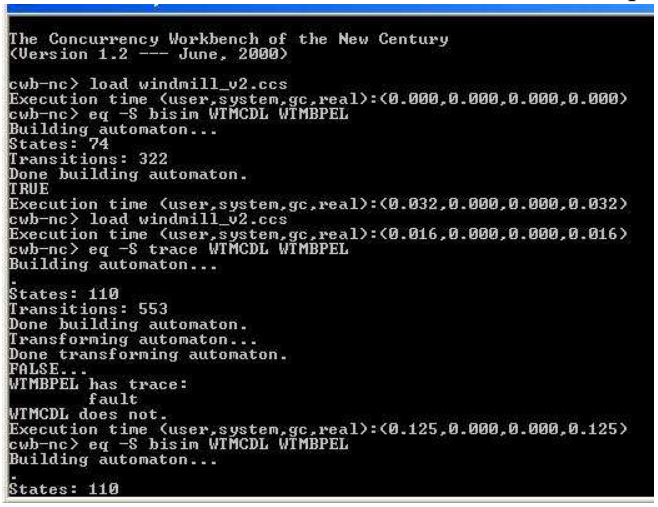
Processes WMC, DMC, and PMC correspond to windturbine management system, demand management system and productivity management system respectively as modeled in Figure 5. The three processes communicate through synchronization events. For instance, *request_n_t* in windturbine management and productivity management.

The simulation results

Figure 6 shows the result of bisimilarity check between CDL and BPEL. The first check, `eq -S bisim WTMCDL WTMBPEL` checks that they are bisimilar. The system has 74 states and 322 transitions. The CWB-NC reports that the processes are bisimilar as well as trace equivalent as shown in Figure 6 A. Recall that the fault handling events are hidden. Hence the bisimilarity. However, when the fault handler is made part of the system, the CWB-NC reports as expected that they are not trace equivalent. The lower part of Figure 6 B shows this result of checking that the two processes are trace equivalent. It shows that the result is $FALSE$ with an additional information that WTMBPEL has trace: $fault$ while WTMCDL does not. Therefore we note that CDL can only be consistent with an abstract version of BPEL where fault handlers are hidden.



A. WTMCDL and WTMBPEL are trace equivalent



B. WTMCDL and WTMBPEL are trace bisimilar but not with fault handling

Figure 6: Consistency Checking using CWB-NC

5 Comparison with other approaches

Several model checking approaches has been employed to provide some form of analysis. An illustrative example which is well-explained is [Mar05]. It deals with specification in only BPEL where both the abstract model and executable model are specified. The approach is based on Petri nets where a communication graph is generated representing the process's external visible behaviour. It verifies the simulation between concrete and abstract behaviour by comparing the corresponding communication graphs.

Abouzaid and Mullins [AM08] propose a BPEL-based semantics for a new specification language based on the π-calculus, which will serve as a reverse mapping to the π-calculus based semantics introduced by Lucchi and Mazzara [LM07b]. The mapping in this work is implemented in a tool integrating the toolkit HAL and generating BPEL code from a specification given in the BP-calculus. Unlike in our approach, this work covers the verification of

BPEL specifications through the mappings while the consistency of the new language and the generated BPEL code is yet to be considered. As a future work, the authors plan to investigate a two way mapping. We expect that our approach will be useful in this setting by taking care of the consistency part of their approach.

In [KNT06] the authors have presented an approach different from model checking: a state propagation approach. It uses preconditions and post-conditions, and computes weakest execution states. The authors argue that descriptions of preconditions and postconditions are easier and more intuitive compared to linear temporal logic formulae for example. However, similar to the above mentioned approaches, only one language is considered. In this case, consistency checking of Web service function invocations using OWL-S metadata descriptions.

Compared to our approach, the final goal is similar: that is checking of consistency. However, there are some differences in the approach. First, our approach considers more than one language. This is because CDL has a more detailed capture of abstract processes compared to the BPEL abstract processes. Further, BPEL is a programming language to specify the behavior of a participant in a choreography whereas choreography is concerned with describing the message interchanges between participants. In addition, a choreography definition can be used at design time by a participant to verify that its internal processes will enable it to participate appropriately in the choreography. With this, certain properties of individual services can be verified as well as verifying the consistency between the protocols in both BPEL and CDL. This can also be extended with some domain specific languages.

6 Conclusion

We have presented an approach for the analysis of web service contracts which uses model checking as its prime tool. The analysis is kept manageable by separating contract aspects and analyzing them individually. The price we pay for this aspect oriented analysis is a check for consistency between the individually derived models. However, this check by setting up a bi-simulation between automata can perhaps be automated, because the configurations of the two automata are systematically related through naming conventions and similarities in the WS-CDL and WS-BPEL constructs. The ideas are illustrated with an example specification of a Wind Turbine Management System which consists of three major components (with their services).

In the current contribution, we demonstrate the approach using timed

automata as used in the UppAal tool [BLL+96], but in other contexts [RRMP08] we have experimented with using JML [Lea06] for the functional aspects. We have not touched on verification of timing aspects, although this work was initiated in [DPC+05]. Thus the use of UppAal is to some extent a practical decision. We feel that it is well justified for the kinds of analyses that we discuss, because they are concerned with checking the properties of the service as such. For checking implementation conformance, it may not be ideal, and a translation to JML may be much more useful, in particular since Java may be an underlying implementation language, and JML is a formal specification language tailored to Java. Its basic use is thus the formal specification of the behavior of Java program modules. This direction is, however, not the main line of our investigation. The immediate work facing us is to streamline the tool fragments developed for these experiments, and in particular to make true the claim that the bi-simulation can be integrated in a more automated analysis process. It is well known that model checking has its limits, and investigations are also being done of theorem proving approaches [GORS06] which may be more suitable for full implementation conformance checking.

Acknowledgment

The second author is funded by the Nordunet3 Project "Contract-Oriented Software Development for Internet Services".

Appendix A: WS-CDL Description of the Demand Management system

```
<?xml version="1.0" encoding="UTF-8"?>
<package author="SCTR Group" name="" version="1.0">

<token name="WindTurbineRef" informationType="StringType"/>
<token name="ProductivityRef" informationType="StringType"/>
<token name="DemandRef" informationType="StringType"/>

<roleType name="WindTurbineRoleType">
<description type="description"/>
<behaviour name="WindTurbineBehaviour"/>
</roleType>

<roleType name="ProductivityRoleType">
<description type="description"/>
<behaviour name="ProductivityBehaviour"/>
</roleType>

<roleType name="DemandRoleType">
```

```
<description type="description"/>
<behaviour name="DemandBehaviour"/>
</roleType>

<relationship name="DemandProductivity">
<role type="DemandRoleType"/>
<role type="ProductivityRoleType"/>
</relationship>

<relationship name="ProductivityWindTurbine">
<role type="ProductivityRoleType"/>
<role type="WindTurbineRoleType"/>
</relationship>

<channelType name="Demand2ProductivityChannelType">
<role type="ProductivityRoleType"/>
<reference>
<token name="ProductivityRef"/>
</reference>
</channelType>

<channelType name="Productivity2WindTurbineChannelType">
<role type="WindTurbineRoleType"/>
<reference>
<token name="WindTurbineRef"/>
</reference>
</channelType>

<choreography>
<relationship type="DemandProductivity"/>
<relationship type="ProductivityWindTurbine"/>

<variableDefinitions>
<variable name="Demand2ProductivityChannel"
  channelType="Demand2ProductivityChannelType"/>
<variable name="Productivity2WindTurbineChannel"
  channelType="Productivity2WindTurbineChannelType"/>

<variable name="Available" informationType="xsd:boolean"
  roleTypes="Productivity"/>
<variable name="WindTurbineClock"
  informationType="tns:Clock" roleTypes="WindTurbine"/>
<variable name="DemandClock" informationType="tns:Clock"
  roleTypes="Demand"/>
<variable name="ProductivityClock"
  informationType="tns:Clock" roleTypes="Productivity"/>
<variable name="detectedincreaseDone"
   informationType="tns:boolean" roleTypes="Demand"/>
</variableDefinitions>

<assign roleType="Productivity">
<copy name="Available_assign">
<source expression="true"/>
<target variable="Available"/>
</copy>
</assign>

<assign roleType="Demand">
<copy name="detectedincrease">
<source expression="false"/>
<target variable="detectedincreaseDone"/>
</copy>
```

```
</assign>

<sequence>
  <workunit name="demand increase detected"
    guard="cdl:equal(
       cdl:getVariable('tns:DemandClock'),
          '','),'0:00')" block="true">
  <assign roleType="DemandRoleType">
  <copy name="calculateincrease"
        causeException="true">
   <source variable="true"/>
   <target variable=
     "cdl:getVariable('detectedincreaseDone',
        '','')"/>
  </copy>
  </assign>
</workunit>

  <interaction name="Demand management system"
    operation="sendIncreasing"
    channelVariable="Demand2ProductivityChannel">
   <participate relationshipType="DemandProductivity"
     fromRole="DemandRoleType"
     toRole="ProductivityRoleType"/>
   <exchange name="CalculatedIncreasing" action="request"/>
   <timeout time-to-complete= "cdl:minor(cdl:getVariable
       ('tns:DemandClock', '','),'0:01')"/>
  </interaction>

  <interaction name="RequestTurbines_interaction"
    operation="RequestTurbines"
    channelVariable="Productivity2WindTurbineChannel">
  <participate
    relationshipType="ProductivityWindTurbine"
    fromRole="ProductivityRoleType"
    toRole="WindTurbineRoleType"/>
  <exchange name="RequestTurbinesExchange" action="request"/>
  <timeout time-to-complete= "cdl:minor(cdl:getVariable
      ('tns:ProductivityClock','',''), '0:02')"/>
  </interaction>

  <interaction name="AvailableTurbines_interaction"
    operation="AvailableTurbines"
    channelVariable="Productivity2WindTurbineChannel">
  <participate
    relationshipType="WindTurbineProductivity"
    fromRole="WindTurbineRoleType"
    toRole="ProductivityRoleType"/>
  <exchange name="AvailableTurbinesExchange" action="request"/>
  </interaction>

  <choice>
    <workunit name="alt_else1_if"
      guard="Available == true" block="true">
      <interaction name="TurbinesOn_interaction"
        operation="TurbinesOn"
        channelVariable="Productivity2WindTurbineChannel">
      <participate
        relationshipType="ProductivityWindTurbine"
        fromRole="ProductivityRoleType"
        toRole="WindTurbineRoleType"/>
      <exchange name="TurbinesOnExchange"
```

```
        action="request"/>
      </interaction>
    </workunit>
    <workunit name="alt_else1_else"
      guard="Available != true" block="true">
      <interaction name="Imposible_interaction"
        operation="Impossible"
        channelVariable="Demand2ProductivityChannel">
      <participate relationshipType="ProductivityDemand"
        fromRole="ProductivityRoleType"
        toRole="DemandRoleType"/>
      <exchange name="ImposibleExchange"
        action="request"/>
      </interaction>
    </workunit>
  </choice>
</sequence>
</choreography>
</package>
```

Appendix B: CCS Description of the Wind Mill Management system in CDL and BPEL

```
***********************************************
*  This models the Wind Mill Management System
*
*  CDL system is consistent with abstract BPEL
*
***********************************************

****  CDL Specification Description  ***********

proc WTMCDL = (WMC | DMC | PMC)\
{request_n_t, available_t,
noavailable, available,
increase_demand, unattended, performsI}
***********************************************

proc WMC =
request_n_t.'available_t.('noavailable.WMC
+ 'available.WMC)

proc PMC =
increase_demand.'request_n_t.available_t.
(available.'performsI.PMC
+ noavailable.'unattended.PMC)

proc DMC =
increase_demand.(unattended.DMC + performsI.DMC)

************************ BPEL  ******************

proc WTMBPEL = (WMC | DMC | PMC | FH)\{fault, reset
request_n_t, available_t,
noavailable, available,
increase_demand, unattended, performsI}
***************************************************
proc FH = fault.'reset.FH
```

```
proc WMB =
request_n_t.('novalue.WMB + 'available_t.
('noavailable.WMB + 'available.turbines_on.WMB))

proc PMB =
increase_demand.'request_n_t.
('reset.PMB + (available_t.
(available.'turbines_on.'performsI.PMB
+ noavailable.'unattended.PMB)))

proc DMB =
increase_demand.('reset.DMB +
(unattended.DMB + performsI.DMB))
```

Paper C: Analyzing Web Service Contracts - an aspect oriented approach[3]

Authors:
M. Emilia Cambronero
University of Castilla-La Mancha, Spain

Joseph C. Okika and Anders P. Ravn
Aalborg University, Denmark

Abstract

Web services should be dependable, because businesses rely on them. For that purpose the Service Oriented Architecture has standardized specifications at a syntactical level. In this paper, we demonstrate how such specifications are used to derive semantic models in the form of (timed) automata. These can be used to model check functional and behavioural properties of a given service. Since there might be several specifications dealing with different aspects, one must also check that these automata are consistent, where we propose to set up a suitable simulation relation. The proposed techniques are illustrated with a small case study.

[3]This chapter is previously published in [COR07].

1 Introduction

The interest in web services has grown in recent times as more and more intra/inter-organizational applications use this model. Thus there is consensus today, that a web service is a programmable component that provides a service and is accessible over the Internet. They are based on standards like SOAP [LM07a, See01, KGH+07], can be standalone, or linked together to provide enhanced functionality. Managing Aero-Electric Wind Turbines, buying airline tickets, accessing an on-line calendar, and obtaining tracking information for your shipments are all business functions that are implemented as web services.

Businesses depend on web services, therefore their properties are of great importance, and informal checking and consensus approaches to when a service is good enough may not suffice. A business will only reluctantly use the offered enterprise applications, because of the high risks involved in using untrusted services from unknown providers. Formal contracts defining the desired properties are therefore studied intensively today, because they are a way to manage the risks that come with the interaction among these inter-organizational services.

Traditionally, contracts in an object oriented setting consider only the functional aspect (pre-condition, post-condition, invariant) of an interface specification. A pre-condition is a constraint than must be satisfied before calling a method or operation; it checks for valid arguments. A post-condition is a corresponding property that is true when the call completes; it is the input-output relation. Finally, an invariant is a constraint on the state of an object; it must hold before or after any operation, and clearly after initialization of the object. These concepts, as popularized by Meyer's "Design by Contract" [Mey97], are, however, part of the complex picture for web services. Since web services are intrinsically distributed, they are by nature concurrent programs, and thus their overall functionality depends not only on correct implementation of the local functionality by sequential algorithms, but even more on the interplay between local functionality and global behavior (protocols and timing).

With SOA it is possible to have a detailed and standardized contract specification as found in WS-BPEL [ACD+03] and WS-CDL [KBR+04]. That leaves the interesting question of how the contracts are used in web service development. Everyone can agree that the contracts have to be satisfied by all the parties involved, but this means that there should be a possibility to take a contract for a web service, and

- check implementations for conformance,

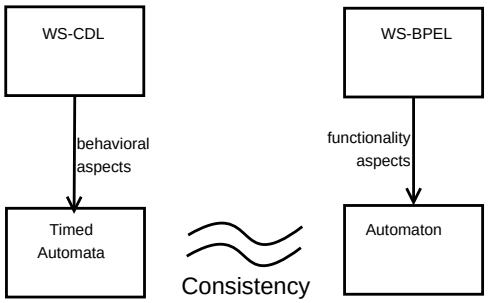


Figure 7: Analysis of Web Service Contracts

- and analyze the contract for consistency.

The contribution of this paper is a solution to these questions following the approach in Figure 7 by

- a translation of the behavioral aspects of a contract to a timed automata for model checking,
- a translation of the functionality to another automaton,
- a check for the consistency of the two automata.

The approach thus covers two major aspects of contracts, and the approach lends itself to generalization to further quantitative aspects, e.g. performance analysis with queuing models. Here, performance would be analyzed with model, and consistency checked with the other models.

Overview

In Section 2, we give a detailed presentation of Web Service contracts where the aspects of contracts are described. We introduce in this section, a case study of an Wind Turbine Management System. Section 3 details the analysis of Web Service contracts. General consistency, satisfiability, consistency and application specific issues are presented. A discussion of related work follows and finally, we conclude in Section 5.

2 Web Service Contracts

In order to come up with a web service contract specification, different levels of a contract are considered:

Contract Levels/Aspects

From an Object Oriented view, an interface describes various methods supported by an object and those which can be invoked by other objects. An Interface Definition Language (IDL) allows the definition of objects based on their interfaces without been concerned with how those interfaces are implemented. Conventionally, interface definitions specify modules and their interface names and operation signatures. Thus, the interface definitions are the contract.

In a service-oriented view, functionality is defined through services which can send and receive messages. With this, applications are composed by combining services that interact through message exchanges. This is done by forming a contract which the interacting services must agree on. Thus, a contract is a specification of the way a consumer of a service will interact with the service provider. A service contract specify functional, behavioral and other aspects. A contract thus defines a runtime dependency between the provider and the consumer.

None of the present contract frameworks combine both the functional, behavioral and QoS aspects, or say much about how the properties should be analyzed. However, we have focused on two of the more popular approaches: Business Process Execution Language (WS-BPEL) and the Web Services Choreography Description Language (WS-CDL).

They are typical and as illustrated in the case study below, they specify different aspects.

Example

We describe in this subsection, a case study of a Wind Turbine Management System. The system monitors and controls wind turbines, and it has several components which could be web services located in different places. We focus on three of these components, because it gives us the scenario needed to specify a web service contract. The components are briefly described below and shown as an UML component diagram in Figure 8.

- Wind Turbine Management: sends a report to Productivity management every hour.
- Productivity Management: receives and analyzes the report from Wind Turbine Management.
- Demand Management: generates a report of power needs for Productivity Management.

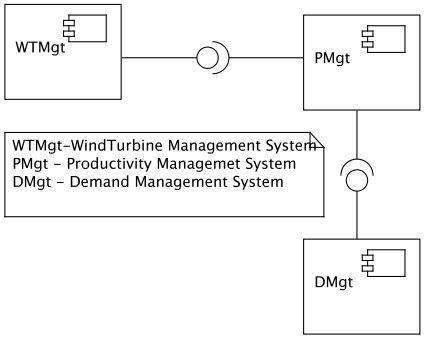


Figure 8: Wind Turbine Management System Components

We look at this example from two perspectives; WS-CDL and WS-PBEL. WS-CDL provides a definition of the information formats being exchanged by all participants. In other words, it specifies the protocols. WS-BPEL provides the message exchanges as viewed by one participant. It describes a business process at a high level.

Contract Aspects in WS-CDL

CDL offers a model for specifying a common understanding of message exchanges. The key aspects of contracts in WS-CDL is itemized below accompanied by the syntax.

- Interface: WSDL defines where it is possible to find different kinds of interface types depending on the specific required web services *interface*.

  In WS-CDL, each interface is associated with a particular role, the syntax is the following:

```
<roleType name="NCName">
   <behavior name="NCName" interface="QName"? />+
</roleType>
```

- Functional Specification: pre-conditions, post-conditions and invariants.

  In WS-CDL these elements are defined by means of *workunits*; which are guarded activities.

```
<workunit name= "demand increase detected"
    guard="cdl:equal(cdl:getVariable
        ('tns:Clock1','',''),'0:00')"
       block="true">
   <assign roleType="tns:DemandRoleType">
    <copy name="calculateincrease"
                causeException=''true">
        <source variable="true"/>
        <target variable="cdl:getVariable
          ('detectedincreaseDone','','')"/>
    </copy>
   </assign>
</workunit>
```

The *workunit*'s *guard* element establishes the condition, which has to be fulfilled to perform the workunit activities, this element allows us to define pre-conditions. Postconditions and invariants can be introduced by appending a workunit with the condition as a guard at the end of the normal workunit flow. In order to define a condition we use XPath and XML Schema expressions.

- Protocol: sequence, non-deterministic choice, external choice, iterations.
  - A sequence of activities is modeled in WS-CDL using the ordering structure *sequence*.
  - A non-deterministic choice is implemented in WS-CDL using the ordering structure *choice*. The WS-CDL standard says that when two or more activities are specified here, only one of these is selected and the other ones are disabled. It is assumed that the selection criteria for those activities are non-observable.

    ```
    <choice>
         Activity-Notation+
    </choice>
    ```

  - External choice. This element can be implemented in WS-CDL using the ordering structure *workunit*, since it allows us to establish conditions to execute the corresponding activity.
  - Iteration. In WS-CDL, we can furthermore use the workunit *repeat* to implement repetition.

- Time aspects: lower bounds, upper bounds, explicit clocks, reset and stop operations are handled by using XPath and XML Schema. Specifically, we use the XML Schema notation to specify the time aspects as follows:

– Explicit clocks are introduced by *xs:time*.

– Lower and upper bounds are specified inside a workunit guard, where XML Schema uses two operations to delimit the time: *op:time-less-than* and *op:time-greater-than*. We can also use the *hasDeadlinePassed* operation, which is defined in the WS-CDL specification to manage timing.

– Reset. In WS-CDL we reset the clock using an *assign* activity, which creates or changes the variable defined by the target element using the expression defined by the source element (in the same role).

– Stop. In order to model that a clock is stopped, we can capture the value of the time, of this specific instant, in a clock variable and then, when we want to initiate the time again, we can use the clock variable to continue from this point. We use two *assign* activities to capture and change the time value.

– Synchronizations. The *interaction* WS-CDL element defines how the parties in a web services are synchronized. The *optional* exchange element allows information to be exchanged during an interaction. The attribute name is used to specify a name for this exchange element.

```
<interation name="The demand management system
             sends increase in power demand to
             the productivity system"
  operation= = "sendIncreasing"
   channelVariable="Demand2ProductivityC">
   <description type="description">
    Sending the necessary increase of demand
   </description>
  <participate
         relationshipType= "DemandProductivity"
         fromRole="DemandRoleType"
         toRole="ProductivityRoleType" />
  <exchange name= "CalculatedIncerasing"
     informationType="Increase_demandType"
        action="request">
  </exchange>
  <timeout
    time-to-complete= "cdl:minor(cdl:getVariable
     ('tns:Clock1','',''),'1:00')">?
</interaction>
```

Contract Aspects in WS-BPEL

BPEL is a programming language to specify the behavior of a participant in a choreography. Choreography is concerned with describing the message

interchanges between participants. In like manner as in WS-CDL, we present the WS-BPEL contract aspects below:

- Interface. In WS-BPEL, the services with which a business process interacts are modeled as *partnerLinks*. Each *partnerLink* is characterized by a *partnerLinkType*, which characterizes the conversational relationship between two services. It defines the roles played by each of the services in the conversation and specifies the *portType* provided by each service to receive messages within the context of the conversation. These *portTypes* are defined in the WSDL document, and each role specifies exactly one WSDL *portType*.

  In order to utilize operations via a *partnerLink*, the binding and communication data, including *endpoint references (EPR)*, for the *partnerLink* must be available. The fundamental use of endpoint references is to serve as the mechanism for dynamic communication of port-specific data for services.

  An example fragment of a *partnerLink* is:

```
<partnerLinks>
<partnerLink name="productivity">
partnerLinkType="as:productivityDemandMSLT"
   myRole="DemandMS"
   partnerRole="productivity" />
</partnerLinks>
```

  The endpoint references syntax is:

```
<service-ref reference-scheme="http://example.org">
  <foo:barEPR xmlns:foo="http://example.org">
   ... </foo:barEPR>
</service-ref>
```

- Functional Specification: preconditions, postconditions and invariants.

  WS-BPEL uses expressions to implement the functional part of a web service contract. WS-BPEL uses several types of expressions, as follows:

  - Boolean expressions. These expressions can appear inside a transition, a join, a while, and an if condition.
  - Deadline expressions. The WS-BPEL elements that use these expressions are until-expressions of onAlarm and wait.
  - Duration expressions. These appear in the *for* expression of *onAlarm* and *wait*, and the *repeatEvery* expression of *onAlarm*.

  - Unsigned Integer expressions combined with *startCounterValue*, *finalCounterValue*, and the branches in a *forEach*.
  - General expressions inside assign activities.

- Protocol: sequence, non-deterministic choice, external choice, iterations.

  - A sequence of activities is modeled by the *sequence* structured activity. It contains one or more activities that are performed sequentially, in the lexical order in which they appear.

    An example of an activity in a Productivity process is given as a sequence as follows:

```
<sequence>
 <if
 bpel:getVariableProperty('x','time:level')==0>
   <then>
  <!-Process productivity (invoke) - ->
   <assign>
<copy>
<from partnerLink="productivityMS"
endpointReference="myRole" />
<to>&increaseData.productivityMSRef </to>
</copy>
 </assign>
 <invoke name="increaseDemand"
          partnerLink="productivity"
          portType="as:productivityPT"
          operation="process"
          inputVariable="increaseData">
   <correlations>
   <correlation set="increaseIdentification"
   </correlations>
 </invoke>
 </if>
</sequence>
```

    Both non-deterministic choice and external choice are expressed in WS-BPEL by means of *pick* activities, which waits for the occurrence of an event and then executes the activity associated with that event. When several events occur simultaneously, an implementation dependent choice is made. Thus in an analysis, the choice must be modeled as non-deterministic.

  - Conditional. WS-BPEL contains a conventional conditional statement as well. Some (for instance if) is shown in the code fragment below.

  - Iteration. In WS-BPEL we can use the *while* and the *repeatUntil* activities, to model iteration.

```
<while>
   <condition>
      $numberWindTurbine < 10
   </condition>
   <scope>  ...  </scope>
</while>

<repeatUntil standard-attributes>
    standard-elements
    activity
    <condition expressionLanguage="anyURI"?>
       ... bool-expr ...
    </condition>
</repeatUntil>
```

- Time aspects: lower bounds, upper bounds, explicit clocks, reset and stop operations are specified as in WS-CDL using XPath and XML Schema.
  - Explicit clocks, lower and upper bounds are defined using XML Scheme notations, as explained before.
  - Reset. In WS-BPEL we can reset the clock using an *assign* activity, which copies data from one variable to another.

```
<assign validate="yes|no"? standard-attributes>
   standard-elements
     (<copy keepSrcElementName="yes|no"?>
     from-spec   to-spec  </copy> |
      <extensibleAssign>
        ...assign-element-of-other-namespace...
      </extensibleAssign>) +
</assign>
```

  - Stop. In order to model that a clock is stopped in WS-BPEL we do as in WS-CDL.
  - Synchronizations are implemented in WS-BPEL using a *flow* activity, which provides concurrency and synchronization. A *flow* completes when all of the activities enclosed by it have completed.

```
<flow standard-attributes>
    standard-elements
    <links>?
        <link name="NCName">+
</links>
        activity+
</flow>
```

3 Analyzing Web Service Contracts

Having described all the elements of specifications, we now present the translation to automata. In order to perform this translation, we note that WS-CDL and WS-BPEL are XML based languages for describing Web Services.

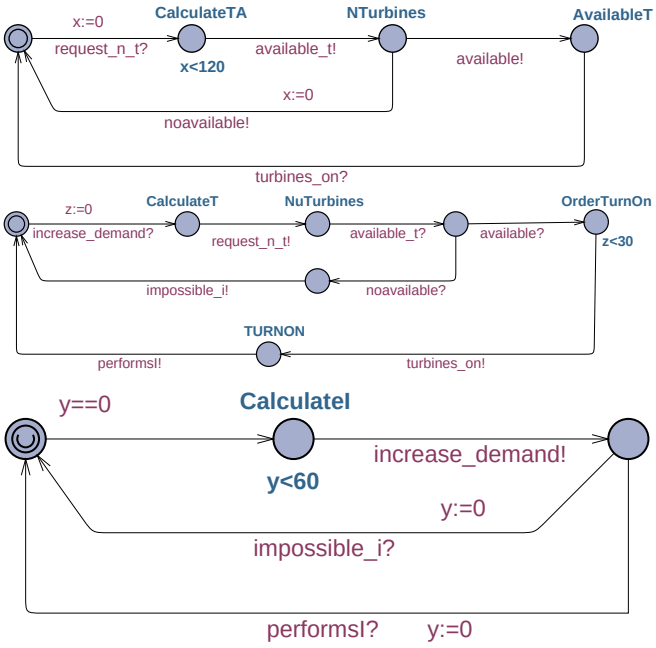


Figure 9: Aero-Electric Management System modeled in UppAal

The timed automata formalism we use is UppAal [BLL+96]; and it is represented by another XML document, thus, the translation has been developed with XSLT [Cla98], XML Style sheets Language for Transformation, which is a language for transforming XML documents into other XML documents.

Figure 9 shows how the translation works: we have created some XSL style sheets, where we use XSLT instructions to extract the information from the WS-CDL document, and then the UppAal document is automatically generated. This document can be opened with the UppAal tool, and thus, we can use the model-checker of UppAal to verify some properties of interest. The tool can also run simulations of the model. We have also created some XSL style sheets to perform the same translation for WS-BPEL documents.

For the two aspects we can check the following.

General Properties

We check the absence of deadlock for the CDL and for the BPEL, we check the property that the system is able to progress from start to termination; in UppAal:

$A[]not\ deadlock$

If there are enough available turbines to fulfill the increase of demand, then the Productivity Management system will send the order of turning on some

of them to the Wind Turbine management system:

$A[]\ WindTurbineMS.AvailableT \rightarrow$
$ProductivityMS.OrderTurnOn$

Satisfiability

Here we check for a BPEL property that the methods can be executed satisfying the contracts or generating the exceptions. For instance, if the demand system always send a message to the productivity system, when it detects an increase in the power demand (the message increase_demand). Also, the Wind Turbine Management system always sends the number of available turbines on Productivity Management system's demand.

This is represented in UppAal as follows:

$A[]\ ProductivityMS.NuTurbines \rightarrow$
$WindTurbineMS.CalculateTA$

Overall Consistency

To check that the two individually derived models are consistent, we use bi-simulation. A bi-simulation is an equivalence relation between state transition systems, associating systems which behave in the same way in the sense that one system simulates the other and vice-versa. The automata generated from the two contract aspects specification systems (WS-CDL, WS-BPEL) are bi-similar in the following aspects:

- they both accept the same operation sequence; since the WS-CDL specified the protocols, while WS-BPEL contains the operation names but with more information.
- they also accept the same message sequence. Thus, the state that receives the message (for instance *increase_demand* in the example in Figure 9) is followed by a state that sends the message (*request_n_t*) in both automata. The automaton from WS-BPEL may contain some internal states.

Application specific

This form of checking is closer to systematic analysis of a design by some review process, for instance Software Reviews, Code Inspections, and other proactive management processes whose purpose is to eliminate or to find and remove errors in product design as early as possible.

4 Related work

Web Service contracts is attracting a lot of attention and several researchers propose various approaches and frameworks toward specification and analysis. For instance [CGP07, CCLP06, DKR04] looks at it in a theoretical dimension, whereas [RGWC99, PS07] propose a language for contracts. All these points to the fact that there is an important need for contracts to be specified and analyzed.

An earlier treatment of contracts in an object-oriented paradigm is Design by Contract [Mey97]. Similar treatment concerning components is found in [BJP99]. Here, the functional specification is achieved through assertions; which consists of preconditions, post-conditions and invariants. The framework in [Mey92] takes a pragmatic approach at code level where the assertions are part of the language. We agree that these functional specifications are important in order to specify a formal agreement between a service provider and its clients. Thus expressing what a client should do in order to make a service request and what the provider will do in return.

Among the related work of Web Service contracts is [HL05]. It proposes to visualize contracts by graph transformation rules. Apart from expressing contracts in terms of pre- and post-conditions of operations together with invariants, they introduced the notions of provided and required contracts. With this, they use the provided contracts to create the test cases and test oracles whereas the required interfaces are used to drive the simulation. We like their treatment of functional specifications, but it needs to be supplemented with other aspects, and one may gain something by investigating model checking as a supplement to testing.

A different quantitative aspect is researched in [KL03, WS-04, wsa04]. The Web Service Level Agreement (WSLA) framework [KL03] is targeted at defining and monitoring SLAs for Web Services. WSLA enables service customers and providers to unambiguously define the agreed performance characteristics and the way to evaluate and measure them. We want to mention here that WSLA complements Web Service Definition Language (WSDL) [CCMW01, BL06], an XML grammar that describes the capabilities of Web services through its interface descriptions. It serves as contracts between service provider and service requestor, but its treatment of functional behavior is limited.

A remarking difference between this work and the above mentioned contributions which mainly focus on either only the functional side of a contract or only the behavioral side of a contract is that a multi view (functional, behavioral) of a web service contract and a set of tools are proposed for the analysis while ensuring consistency.

5 Conclusion

We have presented an approach for the analysis of web service contracts which uses model checking as its prime tool. The analysis is kept manageable by separating contract aspects and analyzing them individually. The price we pay for this aspect oriented analysis is a check for consistency between the individually derived models. However, this check by setting up a bi-simulation between automata can perhaps be automated, because the configurations of the two automata are systematically related through naming conventions and similarities in the WS-CDL and WS-BPEL constructs.

In the current contribution, we demonstrate the approach using timed automata as used in the UppAal tool [BLL+96], but in other contexts [RRMP08] we have experimented with using JML [Lea06] for the functional aspects. The ideas are illustrated with an example specification of a Wind Turbine Management System which consists of three major components (with their services).

We have not touched on verification of timing aspects, although this initiated this work [DPC+05, DCP+06]. Thus the use of UppAal is to some extent a practical decision. We feel that it is well justified for the kinds of analyses that we discuss, because they are concerned with checking the properties of the service as such. For checking implementation conformance, it may not be ideal, and a translation to JML may be much more useful, in particular since Java may be an underlying implementation language, and JML is a formal specification language tailored to Java.

Its basic use is thus the formal specification of the behavior of Java program modules. This direction is, however, not the main line of our investigation. The immediate work facing us is to streamline the tool fragments developed for these experiments, and in particular to make true the claim that the bi-simulation can be integrated in a more automated analysis process. It is well known that model checking has its limits, and investigations are also being done of theorem proving approaches [GORS06] which may be more suitable for implementation conformance checking.

Acknowledgment

The second author is funded by the Nordunet3 Project "Contract-Oriented Software Development for Internet Services".

Paper D: On the Specification of Full Contracts[4]

Authors:

Stephen Fenech, Gordon J. Pace, Joseph C. Okika, Anders P. Ravn and Gerardo Schneider

Abstract

Contracts specify properties of an interface to a software component. We consider the problem of defining a full contract that specifies not only the normal behaviour, but also special cases and tolerated exceptions. In this paper we focus on the behavioural properties of use cases taken from the Common Component Modelling Example (CoCoME), proposed as a benchmark to compare different components models. We first give the full specification of the use cases in the deontic-based specification language $\mathcal{CL}$, and then we concentrate on three particular properties in order to compare deontic and operational specifications. We conjecture that operational specifications are well suited for normal cases, but are less easily extended for exceptional cases. Logic based specifications are essentially compositional, helping in the specification of exceptional cases. This hypothesis is investigated by comparing specifications in CSP (operational) with specifications in $\mathcal{CL}$. The outcome of the experiment supports the conjecture and demonstrates clear differences in the basic descriptive power of the formalisms.

Keywords: Contracts, CoCoME, deontic specifications, operational specifications

[4]This chapter is previously published in [FPO+09].

1 Introduction

Modern software applications are built from components that are connected either statically or dynamically, for instance using a service oriented architecture for Internet-based applications. Components are developed by different teams that may be distributed across countries and organisations. With this reality, it becomes important that the interfaces and protocols used between components are well specified, that there are some *contracts* that regulate these issues. Here the concepts and techniques developed in the formal methods community attract attention. One example is contracts as functional specifications in terms of invariants, pre- and postconditions which are predicates over state variables and parameters that define an input, pre-state, output, post-state relation. The behaviour of components, i.e. the acceptable sequences of method calls or signals that can be exchanged among components, is also important to understand the overall result of connecting distributed, concurrently executing components for an application, as it is done in a service oriented architecture. Here, operational specifications are quite popular; they include both automata based approaches and language oriented process algebras. Deontic logics have not been used to the same extent, although they would offer greater potentials for abstraction from the actual implementations and give a constraint oriented specification style. To some extent this is understandable, because logic formulae are more abstract and not so easy to understand as models. However, they may have an advantage when it comes to providing a full specification of a contract which includes not only the normal use cases, but also special cases with compensations, tolerance of deviations or faults, or exception handling. Here operational models quickly become complex, because they have to specify the compensations and alternatives by branching to different paths.

In this paper, we start by giving the specification in $\mathcal{CL}$, a deontic-based formal language for contracts [PS07], of a large case study which was developed to compare different formal approaches for the specification and analysis of a component based system of a realistic complexity — the CoCoME (*The Common Component Modelling Example*) experiment [RRMP08]. This case study involves all the usual aspects of functionality and behaviour; but also aspects like performance, timing constraints and even dependability. We then investigate contract specifications using logic ($\mathcal{CL}$) and operational models (CSP [Hoa85]) by looking at a fragment of CoCoMe; and we also contrast these with specifications in LTL and CTL [Pnu77]. Since we want to examine the particular hypothesis about operational versus logic specifications, we limit ourselves to behaviours, where the distinction will come out. We have furthermore isolated a particular component where interaction with hu-

mans and external organisations come to the surface. This is where handling exceptions and exceptional cases becomes important to capture the total behaviour so as to avoid unexpected cases. For example, let us consider part of the informal specification of a supermarket cash desk: "While in express mode (allowing only clients with less than 8 items), if no sale is currently taking place, the cashier can choose to disable the express mode". From the behavioural point of view a sequence of events consisting of clients with more than 8 items coming into an express cashier and the subsequent payment, seems to be acceptable given that the cashier can make an exception. Any specification language whose semantics would accept such a sequence would in principle be considered a suitable formalism. This is, however, only partially correct, since it will depend on which kind of properties we are interested in. Just the sequence of events does not keep the original informal specification which uses expressions like "can choose". This kind of modalities add extra information which may be lost by simply observing the sequence of events.

The contributions of this paper are twofold. First, we formalise the specification of the behavioural aspect of CoCoMe in $\mathcal{CL}$. Second, we take 2 use cases from CoCoME to compare deontic ($\mathcal{CL}$) and operational (CSP) specifications.

The paper is organised as follows. In next section we provide a general description of CoCoME. In section 3 we present the language $\mathcal{CL}$ and we give the CoCoME specification. In section 4 we present in detail the three properties to be specified in section 5 using $\mathcal{CL}$ and CSP, and we briefly comment on the suitability of LTL and CTL as specification languages in this context. We compare the specifications in section 6, to conclude in the last section.

2 CoCoME

The Common Component Modelling Example (CoCoME) [RRMP08] is based on a Trading System that handles the sales and inventory of a Store chain. The case study is defined using 8 use cases that describe the main processes. The use cases span from selling products at a cash desk to the exchange of product between stores. The use cases are described as a sequence of actions that must occur followed by a list of exceptional behaviour if the use case allows such behaviour.

Use Case 1 describes how a sale is processed, from the scanning of the items to the payment, either by cash or card. In the exceptional situation that

a card validation fails, the cashier should retry the validation process or require that the customer pays in cash.

Use Case 2 describes how a cash desk switches to express mode which restricts the total number of items the customer should have.

Use Case 3 describes how products, which are running low, are ordered.

Use Case 4 describes how to receive these orders once the suppliers have delivered the items. In the exceptional situation where the delivery is not correct or complete, the products are sent back to the supplier.

Use Case 5 describes how the system generates stock-related reports.

Use Case 6 describe how the system generates delivery reports.

Use Case 7 describes how the price of a product may be altered.

Use Case 8 describes how products can be exchanged from one store to another when the product is running low in one of the stores. The store running low on a certain product will inform the enterprise server, which will send an update stock request to all 'nearby' stores. With the fresh stock information the enterprise server will decide on which store should exchange the goods and sends the request to send the goods. In the exceptional situation that the enterprise server is unreachable, the request is queued to be retried later. In the exceptional situation that not all the 'nearby' stores reply to the update stock request the enterprise server will wait for 15 min after which it will continue the process assuming that stores that have not responded to the request do not have the required products.

3 Specification of CoCoME using $\mathcal{CL}$ —Use Cases 3-8

In this section we first present $\mathcal{CL}$ [PS07], a language to express contracts as terms over obligations, permissions and prohibitions, and then we show how to specify CoCoME in $\mathcal{CL}$. $\mathcal{CL}$ has the following syntax:

$$
\begin{array}{rcl}
\mathcal{C} & := & \mathcal{C}_O|\mathcal{C}_P|\mathcal{C}_F|\mathcal{C} \wedge \mathcal{C}||[\beta]\mathcal{C}|\langle\beta\rangle\mathcal{C}|\top|\bot \\
\mathcal{C}_O & := & O_C(\alpha)|\mathcal{C}_O \oplus \mathcal{C}_O \\
\mathcal{C}_P & := & P(\alpha)|\mathcal{C}_P \oplus \mathcal{C}_P \\
\mathcal{C}_F & := & F_C(\delta)|\mathcal{C}_F \vee [\alpha]\mathcal{C}_F \\
\alpha & := & 0|1|a|\alpha\&\alpha|\alpha \cdot \alpha|\alpha + \alpha \\
\beta & := & 0|1|a|\beta\&\beta|\beta \cdot \beta|\beta + \beta|\beta^*
\end{array}
$$

This syntax is an extension of that given in [KPS08] where here we add the angle brackets. The semantics of $\mathcal{CL}$ have been given in an extension of μ-calculus, an intuitive explanation of which is given below.

A contract typically consists of two parts: *definitions* ($\mathcal{D}$) and *clauses* ($\mathcal{C}$). We deliberately leave the definitions part underspecified in the syntax above. $\mathcal{D}$ specifies the *assertions* (or conditions) and the atomic actions present in the clauses. In this case, the vocabulary of Table 2. Atomic actions are underspecified, but consist of (at least) three parts: the proper action, the subject performing the action, and the target of (or, the object receiving) the action. Note that, in this way, the parties involved in a contract are directly encoded in the actions.

$\mathcal{C}$ is the general *contract clause*. $\mathcal{C}_O$, $\mathcal{C}_P$, and $\mathcal{C}_F$ denote respectively *obligation*, *permission*, and *prohibition* clauses. $O(\cdot)$, $P(\cdot)$, and $F(\cdot)$, represents the obligation, permission or prohibition of performing a given action. $\wedge$ and $\oplus$ correspond to the classical conjunction and exclusive disjunction, which may be used to combine obligations and permissions. For prohibition clauses $\mathcal{C}_F$, the operator $\vee$ corresponding to disjunction is used. The constraints on which operators may be used to compose which types of clauses are introduced to avoid expressing paradoxical contracts.

The α is a compound action (i.e., an expression containing one or more of the following operators: choice "+"; sequence "$\cdot$", and concurrency "&" — see [KPS08]), while β is a compound action which can also be made up of the Kleene star "*". Note that $\oplus$ cannot appear between prohibitions and $+$ cannot occur under the scope of F.

$\mathcal{CL}$ borrows from propositional dynamic logic [FL77] the syntax $[\alpha]\mathcal{C}$ to represent that after performing α (if it is possible to do so), $\mathcal{C}$ must be satisfied. $\langle\alpha\rangle\mathcal{C}$ captures the idea that the possibility exists of executing α, in which case $\mathcal{C}$ must hold afterwards.

$\mathcal{CL}$ can be extended with the temporal operators $\Diamond$ (*eventually*) and $\Box$ (*always*), with standard semantics [Pnu77]. Thus $\Box\mathcal{C}$ can be defined as $[1^*]C$. Similarly, we can define $\Diamond\mathcal{C}$ (*eventually*) for expressing that $\mathcal{C}$ holds sometime in a future moment, as well as the $\mathcal{U}$ (*until*) and $\bigcirc$ (*next*) operators.

Contrary-to-duty (CTD) contracts, which specify an obligation and reparation contract in case the obligation is not met, is expressed in $\mathcal{CL}$ as $O_C(\alpha)$:

obliging action α, but defaulting to contract C if it not satisfied. Similarly, contrary-to-prohibition (CTP) contracts, specifying a prohibited action α and its reparation clause C in case of violation, can also be expressed: $F_C(\alpha)$.

In what follows we specify CoCoME using the contract language $\mathcal{CL}$. CoCoME specifies both behavioural and functional requirements. $\mathcal{CL}$ does not yet support the specification of timing constraints natively; however, one could encode these constraints in the definition of the actions. We have only done this in cases where the timing constraint affected the behaviour of the system since we are focusing on the behavioural specification. Though $\mathcal{CL}$ is limited when it comes to timing constraints, it will allow us to describe exceptional behaviour easily and concisely.

In this section we will specify use cases three to eight of the CoCoME case study. In the following section we will focus on the most interesting parts of use cases one and two and use them to compare deontic specification with operational specification. In the rest of the paper we will use the action names shown in Table 2. For a more detailed presentation of the $\mathcal{CL}$ specification presented in what follows, refer to [Ste].

Specification of Use Case 3 (Order Products)

1. $\Box$[startOrderProcess]O(listItems&listLowItems)
2. $\Box$[listItems&listLowItems]P(entersAmount)
3. $\Box$[entersAmount]P(mngOrderButton)
4. $\Box$[mngOrderButton]O(placeOrder&displayOrderID)

Once the manager starts the order products process (startOrderProcess) the system is obliged to show the full list of items and the list of items that are running low (listItems&listLowItems). After this the manager has the permission to enter the amount of items he would like to order (entersAmount) after which he is permitted to press the order button (mngOrderButton) in which case the system is obliged to place the order and display the order id (placeOrder&displayOrderID). This use case does not have any exceptional behaviour specified. Furthermore, the distinction between the system *permitting* the manager to do certain actions (e.g. P(entersAmount)) and the system being *obliged* to respond (e.g. O(placeOrder&displayOrderID)) is not explicitly described in the CoCoME specification but rather assumed from the common expectations.

disableExpress	Go to Normal Mode
enableExpress	Go to Express Mode
conditionMet	Condition to go to express mode has been met
startSale	Start a new sale
enterItem	Enter new item
finishSale	Stop entering items and start payment procedure
cashPay	Pay in Cash
cardPay	Pay with Card
correctPin	Pin entered is correct
incorrectPin	Pin entered is incorrect
sendBack	Send customer to another line
> 8	Customer has more than eight items
< 8	Customer has less than eight items
returnItems	Customer forfeits items
startOrderProcess	Manager initiates the start of the Order Products process
listItems	The System lists all the products
listLowItems	The system lists the products which are running out of stock
entersAmount	The store manager chooses the product items to order and enters the corresponding amount
mngOrderButton	The store Manager presses the Order button
placeOrder	The System places the order to the appropriate supplier
displayOrderID	The system displays the order identifier generated to the Store Manager
deliver	Supplyer delivers the ordered stock which is identified by an order ID
completeCorrect	Supplier made a complete and correct delivery. This is checked by the Stock Manager
orderReceived	Manager receives the order by pressing the button Roll in received order
updateInventory	The System updates the inventory
sendBack	The Stock Manager sends the products back to the supplier
enterStoreID	Manager enters the store identifier and presses the button Create Report
displayReport	System displaces a report including all the available stock items in the store.
enterEnterpriseID	Managerenters the enterprise identifier and presses the button Create Report
displayEnterpriseReport	The System generates and displays an Enterprise report
requestOverview	Manager initiates the change price process by requesting the listing of all the available products in the store
listItems	The System lists all the products
selectItem	The Manager Selects an Item
changePrice	The Manager changes price
pressCommit	The Manager commits by pressing enter
commitPriceChange	The System changes the price according to the amount set by the manager
productRunsOut	A product of a store runs out
lowStock	The store server recognises low stock of the product.
productRequest	The Store Server sends a request to the Enterprise Server
inventoryRequest	The enterprise server sends an Inventory request to nearby stores
inventoryReply	The store replies with the inventory information
inventoryUpdate	The enterprise server updates the database and does a database look-up for the product
storeChosen	The enterprise server using an "optimisation criterion" chooses from which store to request the transfer
productReply	The enterprise server sends a message to the receiving store.
transferRequest	The enterprise server sends a message to the transferring store
queueRequest	Store server queues request to enterprise.
15min	15 minutes have passed
allRequestsReceived	All requests have been received

Table 2: Alphabet

Specification of Use Case 4 (Receive Ordered Products)

1. $\Box[\text{deliver}]O_{O(\text{sendBack})}(\text{completeCorrect})$

2. $\Box$[completeCorrect]O(mngOrderButton)

3. $\Box$[mngOrderButton]O(updateInventory)

The case study describes that that Manager is required to check that the supplier has sent the correct and complete order. Instead of defining an action MgrChecksOrder we defined the action completeCorrect since the obligation is on the supplier to send the correct information. Thus here we have that once the delivery is made (deliver) the supplier is obliged to have sent the complete and correct delivery (completeCorrect). If however the supplier has violated this obligation, the manager is obliged to send the order back (sendBack), otherwise he is obliged to process the order (mngOrderButton) and the system is obliged to update accordingly (updateInventory).

Specification of Use Case 5 (Show Stock Report)

1. $\Box$[enterStoreID]O(displayReport)

Once the manager enters the store id (enterStoreID) the system is obliged to display the report (displayReport).

Specification of Use Case 6 (Show Delivery Report)

1. $\Box$[enterStoreID]O(displayReport)

Once the enterprise manager enters the store id (enterStoreID) the system is obliged to display the report (displayReport).

Specification of Use Case 7 (Change Price)

1. $\Box$[requestOverview]O(listItems)

2. $\Box$[listItems]P(selectItem)

3. $\Box$[selectItem]P(changePrice)

4. $\Box$[changePrice]P(pressCommit)

5. $\Box$[pressCommit]O(commitPriceChange)

This use case shows the process of how a manager may change a price of an item. The manager starts this process by requesting a list of available products (requestOverview). The system is obliged to list all the items (listItems)

and give permission to the manager to choose items (selectItem). If the manager does select an item, the system should give permission to the manager to change the price (changePrice) after which it should give permission for the manager to commit the price change (pressCommit). If the manager commits the changes, the system is obliged to make these changes permanent (commitPriceChange).

Specification of Use Case 8 (Product Exchange Among Stores)

1. $\Box$[productRunsOut]O(lowStock)
2. $\Box$[lowStock]$O_{O(\text{queueRequest})}$(productRequest)
3. $\Box$[productRequest]O(inventoryRequest)
4. $\Box$[inventoryRequest]O(inventoryReply)
5. $\Box$[inventoryReply]O(inventoryUpdate)
6. $\Box$[15min + allRequestsReceived]O(storeChosen)
7. $\Box$[storeChosen]O(productReply&transferRequest)

If a product runs out (productRunsOut) the local store server should recognise that this has occurred (lowStock) and is obliged to send a request to the enterprise server(productRequest). If this is not successful (for example the connection is down) then the request should be queued (queueRequest). Once such a request is received by the enterprise server, it is obliged to send an inventory request to all nearby stores (inventoryRequest). Every store that receives this request is obliged to reply with the inventory information (inventoryReply). After every reply the enterprise server updates the local databases (inventoryUpdate). Once the enterprise server receives all the replies from the stores or 15 minutes have passed since the requests were sent (15min + allRequestsReceived) it chooses from where the items should be taken and sends a reply to the original store requesting the items and a message to the store that is going to supply the items (productReply&transferRequest).

4 An Example of a Full Contract –Use Cases 1-2

We shall concentrate on the cash desk part of the example shown in Fig. 10 which have the following constituents:

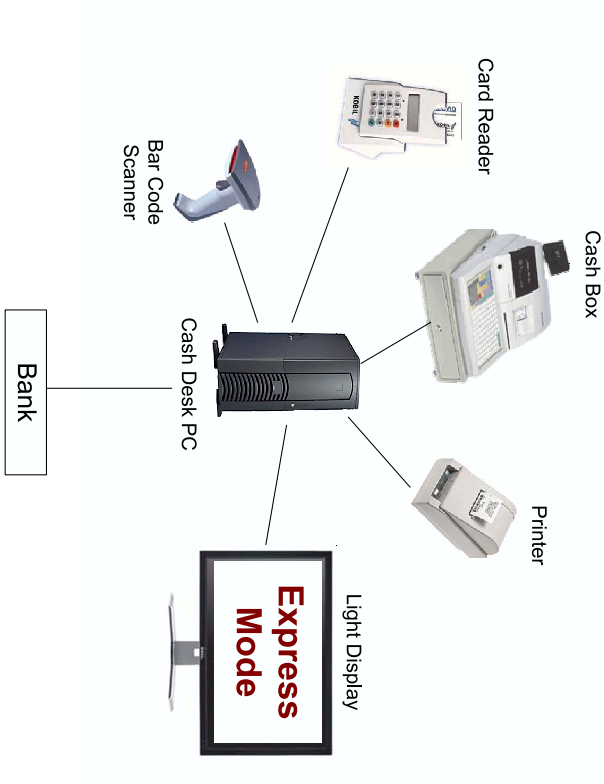


Figure 10: Cash desk and its constituents

(1) Each cash desk has a Cash Box for starting and finishing a sale, and entering received money. (2) In order to identify the products to sell, each cash desk is equipped with a Bar Code Scanner. (3) A Card Reader is installed at each cash desk for handling card payment. Paying by cash can be handled by the Cash Box. (4) In addition there is a Printer for printing the bill which is handed out to the customer at the end of the sale process. (5) To realise the express checkout mentioned above, each cash desk is equipped with a Light Display which signals the customers if the Cash Desk is currently operating in an express mode. If so, the customers are only allowed to buy a small amount of goods and must pay cash in order to keep each transaction short. (6) Each Cash Desk has its own Cash Desk PC where the software handles the sale process, and takes care of the communication with the Bank. Furthermore, it integrates all devices at the Cash Desk.

We focus on the behavioural aspect of the use case, and in particular the following 3 clauses of the contract which includes expected and exceptional behaviours, fairness, permissions and obligations:

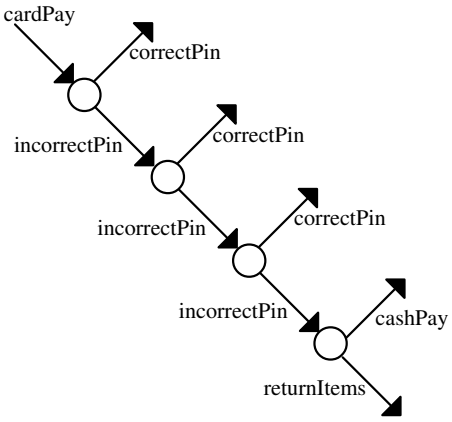


Figure 11: Full transition diagram for cardPay (F1)

F1 If the customer chooses to pay by cash he is obliged to swipe the card followed by entering the correct pin number. If the pin number is incorrect the customer has two more attempts at entering the correct pin after which the client is obliged to pay with cash. If the client refrains to pay with cash the client has to give up the goods. See transition diagram in Fig. 11.

F2 While in normal mode, the cashier may choose to switch to express mode if in the last hour 50% of the sales had less than eight items (conditionMet). Once in express mode the cashier is obliged to eventually go back to normal mode. If conditionMet holds infinitely often, then the cashier should change to express mode infinitely often. See transition diagram in Fig. 12.

F3 In express mode, once a sale has commenced, the cashier is obliged to service customers with less than eight items. To service a customer, the items need to be entered in the system, and then finish the sale. If a customer has more than eight items then it is up to the cashier's discretion whether to service the client or send him back to the end of another line. See Fig. 13.

Clause F2 includes interesting aspects as permissions, obligations and fairness constraints. In Fig. 12 the leftmost state decorated with a black circle indicates that the state should be visited infinitely often. This models the part of the clause which states that the cashier is obliged to always eventually go back to normal mode. From the normal state we can only exit when the express condition is met, after which the cashier has the choice of going back to normal mode or express mode. The dashed transition signifies

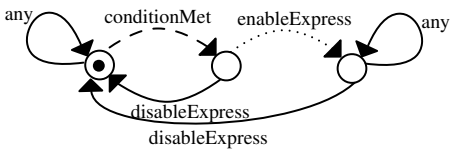


Figure 12: Transition diagram for Express mode (F2)

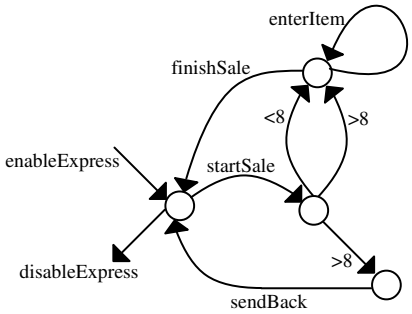


Figure 13: Transition diagram for sales process (F3)

that if this transition is taken infinitely often then the dotted transition needs to be also taken infinitely often, modelling the part of the clause stating that if the condition is met infinitely often then the cash desk needs to infinitely often go into express mode.

In clause F3 the choice to serve a client with more than 8 items is up to the cashier's judgement, This 'permission' to the cashier to 'violate' the rule can be seen as an allowed explicit exception.

5 Formal Specifications of Use Cases 1-2

Our first formal specification is operational, using CSP; it includes the normal operations for the three clauses. Then follows specifications using temporal logics, and finally the deontic logic based specifications. We use the action names shown in Table 2.

Operational Specification

The Relational Calculus of Object and Component Systems (rCOS) is a method for developing component based systems. Syntactically, it is rooted in Unified Theory of Programming (UTP) [HH98] which has been adapted for object and component based use [HLL06a]. Behavioural aspects are syntactically expressed by UML diagrams. Semantically and for verification

—

$$
\begin{array}{lcl}
CashDesk & = & disableExpress \rightarrow NormalDesk \\
 & & [] \ enableExpress \rightarrow ExpressDesk \\
ExpressDesk & = & startSale \rightarrow EnterExp(0) \\
NormalDesk & = & startSale \rightarrow EnterNormal \\
EnterExp(i) & = & i < 8 \quad \wedge \\
 & & enterItem \rightarrow EnterExp(i+1) \\
 & & [] \ finishSale \rightarrow cashPay \rightarrow CashDesk \\
EnterNormal & = & enterItem \rightarrow EnterNormal \\
 & & [] \ finishSale \rightarrow Finish \\
Finish & = & cashPay \rightarrow CashDesk \\
 & & [] \ cardPay \rightarrow CashDesk
\end{array}
$$

Table 3: Normal case specification

purposes, they are translated to CSP [Hoa85].

CSP terms define processes:

$$P \quad ::= \quad Stop \mid a \rightarrow P \mid P[]P \mid P \sqcap P \mid X$$

where *Stop* denotes the deadlocked process; action prefix $a \rightarrow P$ means do a then act as P; external choice ($[]$) between processes, whichever is able to proceed is executed; non-deterministic or internal choice ($\sqcap$), one is chosen; and finally X denotes a process name for a process defined in a set of mutually recursive definitions: $X = P$.

The trace semantics of CSP defines a set of finite traces. For the refusal semantics, which distinguishes the two choice operators, refer to [Hoa85, Ros98].

The Normal Case Specification

The example scenario of sale processing which forms the basis for the example contract is rendered as the CSP processes shown in Table 3. In this specification, we use a bounded integer counter i which ranges from 0 to 8; thus the specification stays within the fragment that can be analysed with a model checker.

Specification of F1

Here we need to modify the $Finish$ process only:

$$\begin{array}{lcl} Finish & = & cashPay \rightarrow CashDesk \ [] \ cardPay \rightarrow Card \\ Card & = & sendPin \rightarrow Check(0) \\ Check(i) & = & correctPin \rightarrow CashDesk \\ & & [] \ i \geq 3 \wedge incorrectPin \rightarrow Nocard \\ & & [] \ i < 3 \wedge incorrectPin \rightarrow Check(i+1) \\ Nocard & = & cashPay \rightarrow CashDesk \ [] \ returnItems \rightarrow CashDesk \end{array}$$

This can be proved to be a refinement of the $Finish$ process in the normal behaviour; but note the intricate branching logic.

Specification of F2

Concerning F2, a non-deterministic switching could be added. It can be specified as follows:

$$Switch \ = \ (enableExpress \rightarrow Switch) \ \sqcap \ (disableExpress \rightarrow Switch)$$

However, there is no guarantee of fairness or liveness, so it is left underspecified.

Specification of F3

Here we have to modify the process $EnterExp$:

$$\begin{array}{lcl} EnterExp(i) & = & (i < 8 \rightarrow enterItem \rightarrow EnterExp(i+1) \\ & & \ [] \ finishSale \rightarrow CashDesk) \\ & & [] \ i \geq 8 \rightarrow Finalise(i) \\ Finalise(i) & = & (finishSale \rightarrow cashPay \rightarrow CashDesk \\ & & \ [] \ enterItem \rightarrow EnterExp(i+1)) \\ & & \sqcap \ finishSale \rightarrow CashDesk \end{array}$$

where $Finalise$ gives the non-deterministic choice of the cashier. Note, however, that in this case we get a process that is no longer a refinement of the previous defined one because it allows same behaviours that were prohibited before.

Temporal Logics Specification

Two widely used temporal logics are LTL and CTL. LTL is a linear temporal logic which allows us to specify properties over paths. Given a set P of atomic prepositions, the syntax of an LTL formula is

$$\varphi \quad ::= \quad p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{G}\varphi \mid \mathbf{F}\varphi \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\varphi$$

The LTL formula $\mathbf{G}\varphi$ means that φ always hold, $\mathbf{F}\varphi$ that φ will eventually hold, $\mathbf{X}\varphi$ that φ will hold in the next step and $\varphi\mathbf{U}\psi$ that φ holds until ψ holds.

CTL is a branching time temporal logic which makes use of the same LTL temporal operators but each temporal operator is preceded by a path quantifier, either **E** or **A**:

$$\begin{aligned}\varphi \quad ::= \quad & p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{AG}\varphi \mid \mathbf{AF}\varphi \mid \mathbf{AX}\varphi \mid \varphi\mathbf{AU}\varphi \mid \\ & \mathbf{EG}\varphi \mid \mathbf{EF}\varphi \mid \mathbf{EX}\varphi \mid \varphi\mathbf{EU}\varphi\end{aligned}$$

E is the existential path quantifier meaning that there exists at least one path starting from this state, which satisfies the quantified formula. **A** is the universal path quantifier meaning that all the paths starting from this state must satisfy the quantified formula.

Specification of F1

The first clause can be seen as a list of conditional statements where it is always the case that after the card is swiped then there is a choice of either entering the correct pin, in which case it would satisfy the formula or else it could be satisfied in the next step. In the next step we repeat the possibility of satisfying the formula by entering the correct pin and if not we again check the next step. This formula can be described in both CTL and LTL:

$$\begin{aligned}\mathbf{AG}(\text{cardPay} \rightarrow \ & \mathbf{AX}\,(\text{correctPin} \ \vee \ \mathbf{AX}(\text{correctPin} \vee \\ & \mathbf{AX}(\text{correctPin} \ \vee \ \mathbf{AX}(\text{cashPay} \ \vee \ \text{returnItems})))))\end{aligned}$$

$$\begin{aligned}\mathbf{G}(\text{cardPay} \rightarrow \ & \mathbf{X}(\text{correctPin} \ \vee \ \mathbf{X}(\text{correctPin} \vee \\ & \mathbf{X}(\text{correctPin} \ \vee \ \mathbf{X}(\text{cashPay} \vee \text{returnItems})))))\end{aligned}$$

Specification of F2

The second clause cannot be described using CTL due to the fairness, unless the logic is extended with fairness constraints. Moreover, it is not clear how the permissions and obligations of the clause could faithfully be represented in CTL. Fairness is expressible using LTL, however, the clause also requires the existence of the transition leading to express mode which cannot be represented using LTL.

Specification of F3

For the third clause it is always the case that once we go to express mode then we need to satisfy the express mode behaviour until we go back to normal mode. Once a sale is started the client needs to be serviced until the sale is finished or the client is sent to another line. If the client has less than eight items then that implies that he should be serviced, otherwise the cashier has to choose between either servicing the customer or sending the customer back. We are also ensuring that there exists the possibility of both servicing the customer and sending the customer back since this is required by the clause. It is because of this requirement that the behaviour cannot be expressed using LTL. However, in CTL it is:

$$
\begin{aligned}
&\mathbf{AG}\ \text{enableExpress} \rightarrow\ \mathbf{AX}(\text{startSale} \rightarrow \\
&\quad \mathbf{AX}((< 8 \rightarrow \mathbf{AX}(\text{enterItem}\mathbf{AU}\text{finishSale})) \wedge \\
&\qquad (> 8 \rightarrow\ \mathbf{AX}(\text{enterItem} \vee \text{sendBack}) \wedge \mathbf{EX}(\text{enterItem}) \wedge \mathbf{EX}(\text{sendBack}) \wedge \\
&\qquad\quad \mathbf{AX}(\text{enterItem} \rightarrow \text{enterItem}\mathbf{AU}\text{finishSale}))) \\
&\quad \mathbf{AU}\ \text{disableExpress})
\end{aligned}
$$

Deontic Specification

In this section we will present a deontic specification of the properties, using $\mathcal{CL}$.

Specification of F1

Here we make extensive use of nested CTDs, where we have a number of options of how the client may satisfy the payment by card. Once a card is swiped then the client is obliged to enter the correct pin (primary obligation). However, if the pin entered is incorrect then the client may still try again two times (secondary obligation) and in case of failure the exceptional cases of paying by cash or returning the items must be enforced. If none is satisfied, the contract is violated:

$$\Box[\text{cardPay}]\ \ O_{\psi_1}(correctPin)$$

where $\psi_1 = O_{\psi_2}(correctPin)$, with $\psi_2 = O_{O(cashPay+returnItems)}(correctPin)$.

Specification of F2

Clause F2 starts by stating that the cashier is infinitely often obliged to go to the normal mode: it can never stay in express mode forever. Then we state that it is always the case that after conditionMet is observed (possibility to

enable the express mode) then the cashier is obliged to either choose to stay in normal mode or express:

$\Box\Diamond O(\text{disableExpress}) \wedge$
$\Box([\text{conditionMet}]\ (O(disableExpress + enableExpress)\ \wedge\ P(enableExpress)\ )) \wedge$
$\Box\Diamond[\text{conditionMet}]\ \Box\ \Diamond O(enableExpress)$

We also enforce that once the condition is met, the cashier has the possibility to go to express mode to avoid a model that only contains a return to normal mode. We do not need to explicitly ensure that there is a possibility to choose to stay in normal mode, similarly to what we have done with the express mode, or that after being in express mode we have the possibility to go back to normal mode because of the first conjunct which states that we have to go infinitely often to normal mode. The fairness requirement is specified in the final part of the clause where we say that if we infinitely often observe conditionMet, then we will infinitely often be obliged to go back into express mode.

Specification of F3

It is always the case that once we go to the express mode a certain behaviour needs to be followed until we go back to normal mode. In the case that the client has less than eight items, then the cashier is obliged to service the customer. However, if the client has more than eight items the cashier is obliged to choose to either service the customer or send back the customer to another cash desk and both possibilities should exist. The last property is thus specified in $\mathcal{CL}$ as follows:

$\Box($ $[enableExpress]($ $[startSale]($
$[< 8]O(enterItem)\,\mathcal{U}$ finishSale $\wedge$
$[> 8](O(enterItem + sendBack)\ \wedge\ P(sendBack)\ \wedge\ P(enterItem) \wedge$
$[enterItem]O(enterItem)\,\mathcal{U}$ finishSale $))$
$\mathcal{U}$ disableExpress $)$

6 Comparison

In Table 4 we present a summary of which formulae can be expressed by the formalisms we used in the previous section. We elaborate in what follows on the differences between the approaches.

The specification of the example using the different notations shows that CTL and CSP allow the specification of exceptional behaviour aspect of

a contract which cannot be specified in other notations such as LTL. Thus making it possible to specify full contracts. However, model based formalisms cannot express global properties such as fairness or liveness of a transition system, because they essentially model the individual transitions.

$\mathcal{CL}$ combines both linear and branching time, with the addition of certain deontic notions. It has not only information of what actions are to be done to satisfy the $\mathcal{CL}$ clause but also prescriptive information about the action, namely whenever the action is observable it is possible to distinguish whether it was required to perform it (as a primary obligation), whether it was a reparation to an obligation, or simply a permitted action.

Moreover, the expression of CTDs and CTPs in terms of basic $\mathcal{CL}$ goes beyond syntactic rewriting, since it still enables a contractual view of when obligations, permissions and prohibitions are active, have been satisfied, or violated. The main advantage of viewing the properties as a deontic contract is that this knowledge is preserved and can be reasoned about.

In summary, F2 seems to be relatively complex property difficult to be captured in specifications using temporal logics and operational approaches. Deontic specifications seem to be appropriate, whenever a right combination of deontic operators with temporal ones is provided.

Analysis Though our aim is to compare the specification style of temporal logics, operational and deontic specifications, we are also interested in what we can do with those specifications, namely how easy it is to analyse them. It is well known that both LTL and CTL are amenable to model checking [CGP99, Hol97]. In the case of rCOS, the analysis of CSP suffices, so one can take advantage of the existing tool FDR2 [Ltd] to do the analysis. The model checker FDR2 may be used to check CSP refinement as well as other properties such as deadlock freeness, trace refinement, etc. However, it is unclear what refinement should be checked for $F3$ since it contains contrary-to-duty actions, which do not blend well with ordinary refinement.

In what concerns $\mathcal{CL}$, an *ad-hoc* algorithm for checking deontic inconsistencies has recently been developed. In this way, given a $\mathcal{CL}$ contract, we are able to detect whether the contract contains contradictory obligations, or an obligation and a prohibition to do something at the same time, and other kinds of contradictions (see [FPS08] for more details). A general model checker for $\mathcal{CL}$ is currently under development, though by using a semantic encoding into an extended μ-calculus [PS07] it is possible to model check contracts written in $\mathcal{CL}$ as presented in [PPS07].

As an additional example to the 3 clauses seen in section 4, let us consider the contract $[a]O(c) \wedge [b]F(c)$ which is satisfiable except when the concur-

rent action $a\&b$ is observed: we end in a state where the contract cannot be satisfied since c is both forbidden and required to happen. We could encode the $\mathcal{CL}$ trace semantics into LTL, however, the correct encoding of the deontic notions as to be able to model check contract inconsistencies would be extremely difficult. Moreover, in order to handle the above small example, CTL and LTL should be extended with concurrent actions, and a priority order among actions (this is built-in in $\mathcal{CL}$ [PS07]).

Summarising, once the specifications are written in any of the approaches under consideration, one can apply existing tools to further analyse them. However, only $\mathcal{CL}$ can be model checked against properties concerning obligations, permissions, and prohibitions, as well as CTDs and CTPs.

	LTL	CTL	CSP	$\mathcal{CL}$
F1	✓	✓	✓	✓
F2	–	–	–	✓
F3	–	✓	(✓)	✓

Table 4: Comparisons between specifications

7 Final Remarks

In this paper we have given a specification of the CoCoME benchmark case study using a deontic specification language. We have then presented and examined the use of three specification styles for the description of total contracts, contracts which not only specify normal behaviours, but also exceptional ones. Clauses of the CoCoME example have been used to illustrate different types of contract clauses and how they can be handled using different specification approaches in order to identify their respective strengths and weaknesses.

One prevailing view of contracts is that of properties which the underlying system must satisfy. In the gist of this view, we have shown how they can be expressed in terms of appropriate standard logics, CTL and LTL. One main disadvantage of this approach is that obligations, permissions and prohibitions are encoded in terms of the underlying logic, making it difficult, in some cases practically impossible, to relate behaviour of the system back to these operators. The encoding also leads to loss of compositionality of contracts for exception handling or contract violations, as in the case of CTDs. Reasoning about CTDS and CTPs would be difficult. In particular,

the detection of deontic inconsistencies, as explained at the end of the previous section, cannot be done in temporal logics, and quite difficult in many operational models.

Using a process calculus approach to describe contracts enables reasoning about the contracts in a direct manner — for instance comparing contracts up to a simulation relation. Also, more complex composition of contracts can be encoded in a direct manner. On the other hand, one still lacks information about contract violation and satisfaction which would have to be encoded directly (and thus prone to error), making the description of total contracts less direct.

Finally, we explore the use of a deontic logic based language to describe the contract clauses. In this approach, we note that reasoning about the deontic state of the system is possible. Moreover, the possibility to analyse contracts, and to express properties of contracts (such as "Whenever you are obliged to pay, you are forbidden from leaving the store, unless you pay") which may refer to the deontic state of the system, is highly desirable. Furthermore, only the analysis of deontic specification is suitable to detect inconsistencies concerning obligations, permissions and prohibitions in full contracts. An implementation of the inconsistency checker for $\mathcal{CL}$ is described in [FPS08].

Overall, it can be argued that the appropriate specification language depends on the intended use. If the contract is intended to be used simply as a property which should be satisfied by a system, then the use of a standard logic, with adequate expressiveness and tool support, will usually suffice. If the use also includes the composition and comparison of contracts, the process calculus approach gives more flexibility. If it is required to analyse and compose full contracts including exceptional behaviour, a deontic approach would be more appropriate.

Paper E: Modelling with Relational Calculus of Object and Component Systems - rCOS[5]

Authors:
Zhenbang Chen, Abdel Hakim Hannousse, Dang Van Hung, Istvan Knoll, Xiaoshan Li, Zhiming Liu, Yang Liu, Qu Nan, Joseph C. Okika, Anders Peter Ravn, Volker Stolz, Lu Yang and Naijun Zhan

Abstract

This chapter presents a formalization of functional and behavioural requirements, and a refinement of requirements to a design for CoCoME using the *Relational Calculus of Object and Component Systems* (rCOS). We give a model of requirements based on an abstraction of the use cases described in Chapter 3.2. Then the refinement calculus of rCOS is used to derive design models corresponding to the top level designs of Chapter 3.4. We demonstrate how rCOS supports modeling different views and their relationships of the system and the separation of concerns in the development.
Keywords: Requirements Modeling, Design, Refinement, and Transformation

[5]This chapter is previously published in [CHH+08].

1 Introduction

The complexity of modern software applications ranging from enterprise to embedded systems is growing. In the development of a system like the CoCoME system, in addition to the design of the application functionality, design of the interactions among the GUI, the controllers of the hardware devices and the application software components is a demanding task. A most effective means to handle complexity is *separation of concerns*, and assurance of correctness is enhanced by *formal modeling* and *formal analysis*.

Separation of concerns Separation of concerns is to divide and conquer. At any stage of the development of a system, the system is divided into a number of views or aspects: the static structural view, the interaction view, the dynamic view and their timing aspects. These views can be modeled separately and their integration forms a model of the whole system. Different concerns require different models and different techniques; state-based static specifications of functionality and their refinement is good for specification and design of the functionality, while event-based techniques are the simplest for designing and analyzing interactions among different components including application software components, GUI components and controllers of hardware devices.

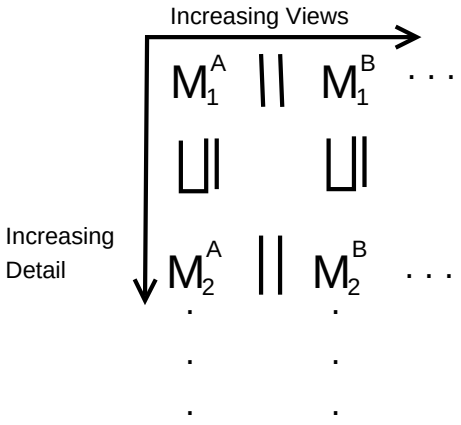


Figure 14: rCOS approach with views and refinements

Formalization In recent UML-based development, the static structural view is modeled by packages of *class diagrams* and/or *component diagrams*, dynamic behavior by state diagrams, and *interactions* by sequence diagrams.

However, UML has a formal syntax only, and its semantics is not formally definable without imposing strong restrictions.

To assure correctness, we need to incorporate semantic reasoning through specification, refinement, verification and analysis into the development process.

To provide formal support to multi-view and multi-level modeling and analysis in a model-driven development process, a desirable method should

1. allow to model different views of the system at different levels of abstraction,

2. provide analysis and verification techniques and tools that assist in showing that the models have the desired properties,

3. give precise definitions of correctness preserving model transformations, and provide effective rules and tool support for these transformations.

Based on these considerations, we have recently developed a refinement calculus, named rCOS, for design of object and component oriented software systems [HLL06a, LLZ06, HLL06b, CHLZ06]. It provides a two dimensional refinement framework, that is, *consistent increments* to the models for the multiple parallel ($\|$) views in the horizontal dimension, and *refinement*, a relation ($\sqsubseteq$) between models at different levels of abstraction in the vertical dimension.

Goals and scope of the component model

The key concepts in the rCOS approach are:

- A component is an aggregation of objects and processes with interfaces. Each interface has a contract with a Hoare style functional specification and a protocol defining the permissible traces of method invocations.

- Components can be composed hierarchically by plugging, renaming, hiding and feedback. These operators are defined by a relational semantics in the style of Hoare and He's Unified Theory of Programming (UTP).

- Composition is supported by required and provided interfaces

- The refinements are verifiable through UTP laws and derived theorems. The compatibility of the compositions in the concrete example have been checked for the protocols using the FDR tool and for the functional specifications, some of them have been checked using JML.

- Application aspects are dealt with in a Requirements Modeling phase. The resulting component specifications are refined to a logical design closer to programs.
- The logical design is refined and refactored to a specific component architecture suitable for a given technical platform.
- The approach does not address deployment but can include implementation code to the extent that the implementation language is given UTP semantics.

Modeled cutout of CoCoME

For the example, we have covered the following:

- The aspects of Requirements Modeling, Logical Design and Component Architecture Design are illustrated. Deployment and testing have not been done.
- The rCOS approach is not currently able to handle dynamic composition, deployment and testing.
- The strong parts of the treatment are formal component specifications, refinement and multi-view consistency checks.
- The weak parts are extra-functional properties and tool support.
- We did not model the exact architecture of the CoCoME because we focused on the systematic refinements step from requirements to a component architecture very similar to the one of CoCoME implementation.
- The protocol and multi-view consistency was verified for the logical design using FDR and JML respectively.

Benefit of the modeling

In a model driven development process, the rCOS approach offers:

- A formal specification and analysis of different views including static structure interactions and functionalities.
- High level refinement rules for adding details in moving from one level of design to another.

- How different tools for verification and analysis of properties may be incorporated.

Effort and lessons learned

Getting acquainted with rCOS requires three months with reasonable background in formal methods in general. In the context of the CoCoME experiment, the current treatment took experienced rCOS people about 2 work months (about 5 people over 1.5 calendar month). We estimate that a complete treatment of the example to the level of a component architecture design would require about 12 work months.

In more detail, one experienced formal rCOS expert spent one day working out the specification of **UC 1**, and the other six use cases took a 4-man week of PhD students, supervised by the experienced person. The OO design of **UC 1** took the rCOS expert less than one hour with writing, but the design of the other use cases took another 4-student week. A lot of effort had to be spent on ensuring model consistency manually. Just from the presented design we have derived around 65 Java classes with over 4000 lines including comments and blank lines. The packages of the *classes* for the resulting component-based model are shown in App. 2.

Among the important lessons from the exercise are that after developing the functional specifications, the other tasks follow almost mechanically. However, there is still much room in both design and implementation for an efficient solution, taking into account for example, knowledge about the uniqueness of keys in lookups resulting from existential quantification.

Overview

In the next section the fundamental concepts and ideas of the rCOS Component Model are introduced. Section 3 shows an rCOS formalization of the requirements for the common example including specifications of classes with local protocols and functional specification. Based on the functional specifications, an object-oriented detailed design is generated. This is refined to components that are fitted into the prescribed architecture of the exercise. Finally we discuss Extra-functional properties and related Tool Support in Section 4. Finally, Section 5 concludes and discusses the perspectives and limitations of the solution. In particular we comment on how extra-functional properties can be included through observables, and likely approaches to extensive tool support.

2 Component Model

We introduce the basic syntax of rCOS that we will use in the case study, with their informal semantics. We will keep the introduction mostly informal and minimal that is enough for the understanding of the rCOS models of CoCoME. For detailed study, we refer the reader to our work in [HLL06a, LLZ06, HLL06b, CHLZ06].

The *Relational Calculus of Object and Component Systems* (rCOS) has its roots in the *Unified Theory of Programming* (UTP) [HH98].

UTP - the background of rCOS UTP is the completion of many years of effort in the formalization of programming language concepts and reasoning techniques for programs by Tony Hoare and He Jifeng. It combines the reasoning power of ordinary predicate calculus with the structuring power of relational algebra. All programs are seen as binary relations on a state space which is a valuation of a set X of program variables or other observables. An example of an observable is a variable ok', which is true exactly when a program terminates. In all generality, the partial relation for a sequential program P is specified by a pair of predicates over the state variables, denoted by $pre(x) \vdash post(x, x')$ and called a *design*, where x represents the values of the variables before the execution of the program and x' denotes the new values for the variables after the execution, $pre(x)$ is the *precondition* defining the domain of the relation, and $post(x, x')$ is the *postcondition* defining the relation itself.

The meaning of the design $pre(x) \vdash post(x, x')$ is defined by the predicate: $pre(x) \wedge ok \Rightarrow ok' \wedge post(x, x')$, asserting that if the program is activated from a well-defined state (i.e. its preceding program terminated) that satisfies the precondition $pre(x)$ then the execution of the program will *terminate* (i.e. $ok' = true$) in a state such that the new values in this state are related with the old values before the execution by *post*. For example, an assignment $x := x + y$ is defined as $true \vdash x' = x + y$.

In UTP, it is known that designs are closed under all the standard programming constructs like sequential composition, choice, and iteration. For example, $D_1; D_2$ is defined to be the relational composition $\exists x_0 : (D_1(x_0/x') \wedge D_2(x_0/x))$.

For concurrency with communicating processes, additional observables are used to record communication traces; communication readiness can be expressed by *guard* predicates. A major virtue of using a relational calculus is that *refinement* between programs is easily defined as relation inclusion or logical implication.

It is clear that UTP needs adaptation to specific programming paradigms,

and rCOS has emerged from the work of He Jifeng and Zhiming Liu to formalize the concepts of object oriented programming: classes, object references, method invocation, subtyping and polymorphism [HLL06a]. They have continued to include concepts of component-based and model-driven development: interfaces, protocols, components, connectors, and coordination [HLL06b, CHLZ06]. Thus rCOS is a solid semantic foundation for component-based design. Also, its refinement calculus has been further developed such that it offers a systematic approach to deriving component based software from specifications of requirements.

With a programming language like Java, OO and component-based refinement can effectively ensure correctness. Most refinement rules have corresponding design patterns and thus Java implementations. This takes refinement from programming in the small to programming in the large.

Object modelling in rCOS

Just as in Java, an OO program in rCOS has *class declarations* and a *main program* [HLL06a]. A class can be public or private and declares its attributes and methods, they can be public, private or protected. The main program is given as a *main class*. Its attributes are the global variables of program and it has a main method *main*(), that implements the application processes. Unlike Java, the definition of a method allows specification statements which use the notion of design $pre \vdash post$. Notice that the use of the variables ok and ok' implies that rCOS is a total correctness model, that is if the precondition holds the execution terminates correctly.

Types and notations

Another difference of rCOS from an OO programming language is that we distinguish data from objects and thus a datum, such as an integer or a boolean value does not have a reference. For the CoCoME case study, we assume the following data types:

$$V ::= long \mid double \mid char \mid string \mid bool$$

Assuming an infinite set CN of symbols, called class names, we define the following type system, where C ranges over CN

$$T ::= V \mid C \mid array[1..n](T) \mid set(T) \mid bag(T)$$

where $array[1:n](T)$ the type of arrays of type T, and $set(T)$ is the type of sets of type T. We assume the operations $add(T\ a)$, $contains(T\ a)$, $delete(T\ a)$

and *sum*() on a set and a bag with their standard semantics. For a variable s of type *set*(T), the specification statement $s.add(a)$ equals $s' = s \cup \{a\}$, $s.sum()$ is the sum of all elements of s, which is assumed to a set of numbers. We use curly brackets $\{e_1, \ldots, e_n\}$ and the square brackets $[[e_1, \ldots, e_m]]$ to define a set and a bag. For set s such that each element has an identifier, $s.find(id)$ denotes the function that returns the element whose identifier equals id if there is one, it returns *null* otherwise. Java provides the implementations of these types via the *Collection* interface. Thus these operations in specifications can be directly coded in Java.

In specifications, $C\ o$ means that object o has type C, and $o \neq null$ means that o is in the object heap if the type of o is a class, and that o is defined if its type is a data type. The shorthand $o \in C$ denotes that $o \neq null$ and its type is C.

In rCOS, evaluation of expressions does not change the state of the system, and thus the Java expression `new` $C()$ is not a rCOS expression. Instead, we take $C.New(C\ x)$ as a command that creates an object of C and assigns it to variable x. The attributes of this object are assigned with the initial values or objects declared in C. If no initial value is declared it will be *null*. However, in the specification of CoCoME, we use $x' = C.New()$ to denote $C.New(x)$, and $x' = C.New[v_1/a_1, \ldots, a_k/v_k]$ to denote the predicate $C.New[v_1/a_1, \ldots, a_k/v_k](x)$ that a new object of class C is created with the attributes a_1 initialized with v_i for $i = 1, \ldots, k$, and this objects is assigned to variable x.

For CoCoME, a *design* $pre \vdash post$ for a method in rCOS is written separately as **Pre** *pre* and **Post** *post*. For the sake of systematic refinement, we write the specification of static functionality of a use case handler class in the following format:

class C [**extends** D] {
attributes $T\ x = d, \ldots, T_k\ x = d$
methods $m(T\ in; V\ return)$ {
pre: $c \vee \ldots \vee c$
post: $\wedge\ (R; \ldots; R) \vee \ldots \vee (R; \ldots; R)$
$\wedge\ \ldots\ldots$
$\wedge\ (R; \ldots; R) \vee \ldots \vee (R; \ldots; R)$ }
$\ldots\ldots$
$m(T\ in; V\ return)$ {$\ldots\ldots$ }
invariant Inv
}

where

- The list of class declarations can be represented as a UML class diagram.

- The initial value of an attribute is optional.
- Each c in the precondition, is a conjunction of primitive predicates.
- Each relation R in the postcondition is of the form $c \wedge (le' = e)$, where c is a boolean condition and le an *assignable expression* and e is an expression. An assignable le is either a primitive variable x, or an attribute name, a, or $le.a$ for an attribute name a. We use **if** c **then** $le' = e_1$ **else** $le' = e_2$ for $c \wedge (le' = e_1) \vee \neg c \wedge (le' = e_2)$ and **if** c **then** $le' = e$ for $c \wedge (le' = e) \vee \neg c \wedge \mathbf{skip}$. Notice here that the expression e does not have to be an executable expression. Instead, e is a logically specified expression, such as the greatest common divisor of two given integers.

  We allow the use of *indexed conjunction* $\forall i \in I : R(i)$ and *indexed disjunctions* $\exists i \in I : R(i)$ for a finite set I. These would be the quantifications if the index set is infinite.

The above format has been influenced by TLA$^+$ [Lam02], UNITY [CM88] and Java. We also need a notation for traces; in this setting, they are given by UML sequence diagrams and the UML state diagrams.

Refinement

In rCOS, we provide three levels of refinement:

1. Refinement of a whole object program. This may involve the change of anything as long as the visible behaviour of the main method is preserved. It is an extension to the notion of data refinement in imperative programming, with a semantic model dealing with object references. In such a refinement, all non-public attributes of objects are treated as local (or internal) variables [HLL06a].
2. Refinement of the class declaration section $Classes_1$ is a refinement of $Classes$ if $Classes_1 \bullet main$ refines $Classes \bullet main$ for all $main$. This means that $Classes_1$ supports at least as many functional services as $Classes$.
3. Refinement of a method of a class. This extends the theory of refinement in imperative programming, with a semantic model dealing with object references. Obviously, $Class_1$ refines $Class$ if the public class names in $Classes$ are all in $Classes_1$ and for each public method of each public class in $Classes$ there is a refined method in the corresponding class of $Classes_1$.

An rCOS design has mainly three kinds of refinement: *Delegation of functionality* (or *responsibility*), *attribute encapsulation*, and *class decomposition*. Interesting results on completeness of the refinement calculus are available in [LLZ06].

Delegation of functionality. Assume that C and C_1 are classes in *Classes*, $C_1\ o$ is an attribute of C and $T\ x$ is an attribute of C_1. Let $m()\{c(o.x', o.x)\}$ be a method of C that directly accesses and/or modifies attribute x of C_1. Then, if all other variables in command c are accessible in C_1, we have that *Classes* is refined by $Classes_1$, where $Classes_1$ is obtained from *Classes* by changing $m()\{c(o.x', o.x)\}$ to $m()\{o.n()\}$ in class C and adding a fresh method $n()\{c[x'/o.x', x/o.x]\}$. This is also called the *expert pattern of responsibility assignment*.

Encapsulation. When we write the specifications of the methods of a class C before designing the interactions between objects, we often need to directly refer to attributes of the classes that are associated with C. Therefore, those attributes are required to be public. After designing the interactions by application of the expert pattern for functionality assignments, the attributes that were directly referred are now only referred locally in their classes. These attributes can then be encapsulated by changing them to protected or private.

The *encapsulation rule* says that if an attribute of a class C is only referred directly in the specification (or code) of methods in C, this attribute can be made a *private attribute*; and it can be made *protected* if it is only directly referred in specifications of methods of C and its subclasses.

Class decomposition. During an OO design, we often need to decompose a class into a number of classes. For example, consider classes $C_1 :: D\ a_1$, $C_2 :: D\ a_2$, and $D :: T_1\ x, T_2\ y$. If methods of C_1 only call a method $D :: m()\{...\}$ that only involves x, and methods of C_2 only call a method $D :: n()\{...\}$ that only involves y, we can decompose D into $D_1 :: T_1\ x; m()\{...\}$ and $D_2 :: T_2\ y; n()\{...\}$, and change the type of a_1 in C_1 to D_1 and the type of a_2 in C_2 to D_2. There are other rules for class decomposition [HLL06a, LLZ06].

With these and other refinement rules in rCOS, we can prove a big-step refinement rule, such as the following **expert pattern**, that will be repeatedly used in the design of CoCoME.

Theorem 1 (Expert Pattern) *Given a class declarations section Classes and its navigation paths* $r_1.....r_f.x$, *(denoted by le as an assignable expression),* $\{a_{11}.....a_{1k_1}.x_1, \ldots, a_{\ell 1}.....a_{\ell k_\ell}.x_\ell\}$, *and* $\{b_{11}.....b_{1j_1}.y_1, \ldots, b_{t1}.....a_{tj_t}.y_t\}$ *starting from class C, let* $m()$ *be a method of C specified as*

$$C :: \quad m()\{ \quad c(a_{11}.\ldots.a_{1k_1}.x_1, \ldots, a_{\ell 1}.\ldots.a_{\ell k_\ell}.x_\ell) \\ \wedge \quad le' = e(b_{11}.\ldots.b_{1s_1}.y_1, \ldots, b_{ts1}.\ldots.b_{ts_t}.y_t) \ \}$$

then Classes can be refined by redefining $m()$ in C and defining the following fresh methods in the corresponding classes:

$$\begin{array}{ll}
C :: & check()\{return'{=}c(a_{11}.get_{\pi_{a_{11}}x_1}(), \ldots, a_{\ell 1}.get_{\pi_{a_{\ell 1}}x_\ell}())\} \\
& m()\{\textbf{if } check() \textbf{ then } r_1.do\text{-}m_{\pi_{r_1}}(b_{11}.get_{\pi_{b_{11}}y_1}(), \\
& \qquad \ldots, b_{s1}.get_{\pi_{b_{s1}}y_s}())\} \\
T(a_{ij}) :: & get_{\pi_{a_{ij}}x_i}()\{return'{=}a_{ij+1}.get_{\pi_{a_{ij+1}}x_i}()\} \ (i : 1..\ell, j : 1..k_i - 1) \\
T(a_{ik_i}) :: & get_{\pi_{a_{ik_i}}x_i}()\{return'{=}x_i\} \ (i : 1..\ell) \\
T(r_i) :: & do\text{-}m_{\pi_{r_i}}(d_{11}, \ldots, d_{s1})\{r_{i+1}.do\text{-}m_{\pi_{r_{i+1}}}(d_{11}, \ldots, d_{s1})\} \\
& for\ i : 1..f - 1 \\
T(r_f) :: & do\text{-}m_{\pi_{r_f}}(d_{11}, \ldots, d_{s1})\{x' = e(d_{11}, \ldots, d_{s1})\} \\
T(b_{ij}) :: & get_{\pi_{b_{ij}}y_i}()\{return'{=}b_{ij+1}.get_{\pi_{b_{ij+1}}y_i}()\} \ (i : 1..t, j : 1..s_i - 1) \\
T(b_{is_i}) :: & get_{\pi_{b_{is_i}}y_i}()\{return'{=}y_i\} \ (i : 1..t)
\end{array}$$

where $T(a)$ is the type name of attribute a and π_{v_i} denotes the remainder of the corresponding navigation path v starting at position j.

If the paths $\{a_{11}.\ldots.a_{1k_1}.x_1, \ldots, a_{\ell 1}.\ldots.a_{\ell k_\ell}.x_\ell\}$ have a common prefix up to a_{1j}, then class C can directly delegate the responsibility of getting the x-attributes and checking the condition to $T(a_{ij})$ via the path $a_{11}\ldots, a_{ij}$ and then follow the above rule from $T(a_{ij})$. The same rule can be applied to the b-navigation paths.

The expert pattern is the most often used refinement rule in OO design. One feature of this rule is that it does not introduce more couplings by associations between classes into the class structure. It also ensures that functional responsibilities are allocated to the appropriate objects that *know* the data needed for the responsibilities assigned to them.

An important point to make here is that the expert pattern and the rule of encapsulation can be implemented by automated model transformations. In general, transformations for structure refinement can be aided by transformations in which changes are made on the structure model, such as the class diagram, with a diagram editing tool and then automatic transformations can be derived for the change in the specification of the functionality and object interactions [LLZ06].

Component modelling in rCOS

There are two kinds of components in rCOS, *service components* (simply called *components*) and *process components* (also simply called *processes*).

Like a service component, a *process component* has an interface declaring its own local state variables and methods, and its behavior is specified by a process contract. Unlike a service component that is passively waiting for a client to call its provided services, a process is active and has its own control on when to call out to required services or to wait for a call to its provided services. For such an active process, we cannot have separate contracts for its provided interface and required interface, because we cannot have separate specifications of outgoing calls and incoming calls. So a process only has an interface and its associated contract (or code).

Compositions for *disjoint union* of components and *plugging* components together, for *gluing components* by processes are defined in rCOS, and their closure properties and the algebraic properties of these compositions are studied [CHLZ06]. Note that an interface can be the union of a number of interfaces. Therefore, in a specification we can write the interfaces separately.

The contracts in rCOS also define the unified semantic model of implementations of interfaces in different programming languages, and thus clearly supports interoperability of components and analysis of the correctness of a component with respect to its interface contract. The theory of refinements of contracts and components in rCOS characterizes component substitutivity, as well as it supports independent development of components.

Related work

The work on rCOS takes place within a large body of work [LE06] on modeling and analysis techniques for object-oriented and component based software. Some of these works we would like to acknowledge.

Eiffel [Mey97] first introduced the idea of design by contract into object-oriented programming. The notion of designs for methods in the object-oriented rCOS is similar to the use of assertions in Eiffel, and thus also support similar techniques for static analysis and testing. JML [Lea06] has recently become a popular language for modeling and analysis of object-oriented designs. It shares similar ideas of using assertions and refinement as behavioral subtype in Eiffel. The strong point of JML is that it is well integrated with Java and comes with parsers and tools for UML like modeling.

In Fractal [PV02], behavior protocols are used to specify interaction behavior of a component. rCOS also uses traces of method invocations and returns to model the interaction protocol of a component with its environment. However, the protocol does not have to be a regular language, although that suffices for the examples in this chapter. Also, for components rCOS separates the protocol of the provided interface methods from that of the required interface methods. This allows better pluggability among compo-

nents. On the other hand, the behavior protocols of components in Fractal are the same for the protocols of coordinators and glue units that are modeled as processes in rCOS. In addition to interaction protocols, rCOS also supports state-based modeling with guards and pre-post conditions. This allows us to carry out stepwise functionality refinement.

We share many ideas with work done in Oldenburg by the group of Olderog on linking CSP-OZ with UML [MORW04] in that a multi-notational modeling language is used for encompassing different views of a system. However, rCOS has taken UTP as its single point of departure and thus avoids some of the complexities of merging existing notations. Yet, their framework has the virtue of well-developed underlying frameworks and tools.

3 The Example

The requirements capture starts with identifying *business processes* described as *use cases*. The use case specification includes four views. One view is the *interactions* between the external environment, modeled as *actors*, and the system. The interaction is described as a protocol in which an actor is allowed to invoke *methods* (also called *use case operations*) provided by the system. In rCOS, we specify such a protocol as a set of *traces* of method invocations, and depict it by a UML *sequence diagram* (cf. Fig. 15), called a *use case sequence diagram*.

In the real use of the system the actors interact with the system via the GUI and hardware devices. However, in the early stage of the design, we abstract from the GUI and the concrete input and output technologies and focus on specifying what the system should produce for output after the input data or signals are received. The design of the GUI and the controllers of the input and output devices is a concern when the application interfaces are clear after the design of the application software. Also, a use case sequence diagram does not show the interactions among the domain objects of the system, as the interactions among those internal objects can only be designed after the specification of what the use case operations do when being invoked. There are many different ways in which the internal objects can interact to realize the same use case.

The interaction trace of a use case is a constraint on the flow of control of the main application program or processes. The flow of control can be modeled by a *guarded state transition system*, that can be depicted by a UML state diagram (cf. Fig. 17). While a sequence diagram focuses on the interactions between the actors and the system, the state diagram is an operational model of the *dynamic behavior* of the use case. They must be

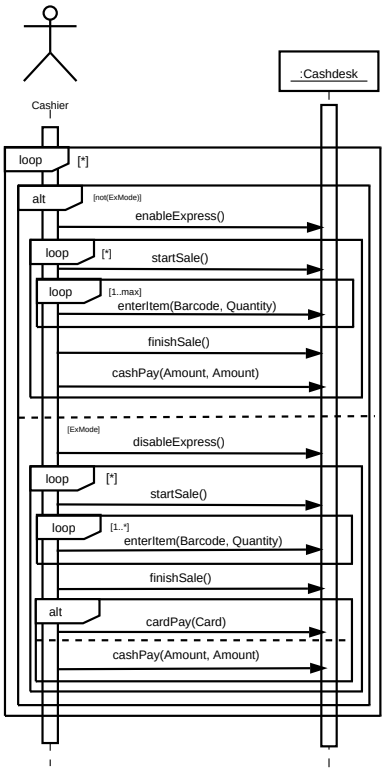


Figure 15: Use case sequence diagram for **UC 1 & 2**

trace equivalent. This model may be used for verification of deadlock and livelock freedom by model checking state reachability.

Another important view is the static *functionality view* of the system. The requirements should precisely specify what each use case operation should do when invoked. That is what state change it should make in terms of what new objects are to be created, what old objects should be destroyed, what links between which objects are established, and what data attributes of which objects are modified. And what is the precondition for carrying out these changes. For the purse of *compositional* and *incremental* specification, we introduce a designated*use case controller class* for each use case, and we specify each method of the use case as a method of this controller class. Each method is specified by its signature and its design in the form $pre \vdash post$. The signatures of the methods must be consistent with those used in the interaction and dynamic views. During the specification of the static functionality of the use case operations, all types and classes (together with their attributes) required in the specification must be defined.

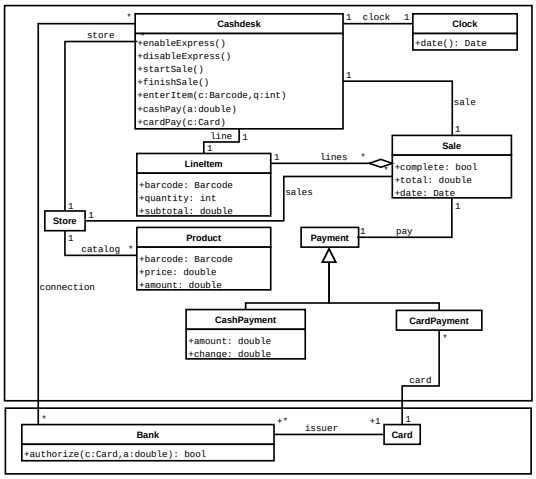


Figure 16: Use case class diagram for **UC 1 & 2**

The type and class definitions in the specification of functionality of the methods of the use case controls forms the *structure view* of the system. It can be depicted by a class diagram or packages of class diagrams (cf. Fig. 16). The consistency and integrated semantics of the different views are studied in [CLM07].

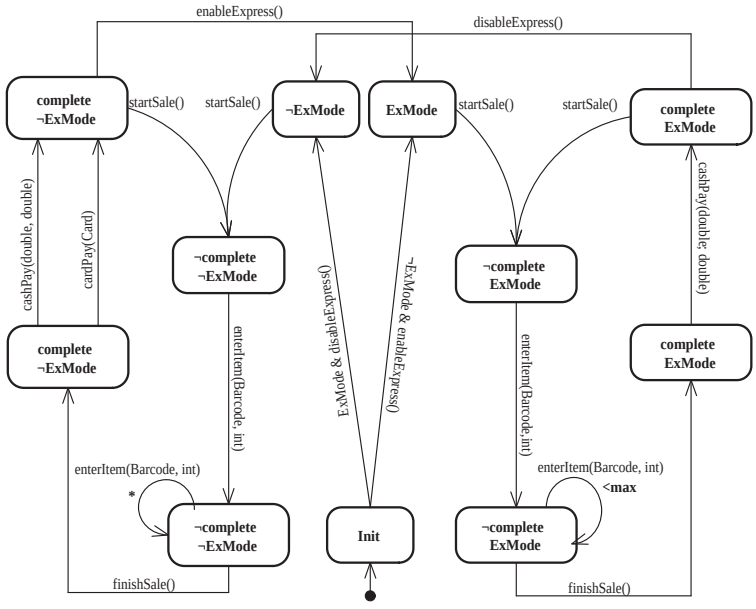


Figure 17: State diagram for **UC 1 and 2**

UC 1 & UC 2: Process sale

As both the first and the second use case relate to a single sale, we handle them in a single section. It would be possible to keep the mode change in a separate use case, but the combination saves space.

We first model the interaction protocol that the system offers the actor, i.e. Cashier. This is given in a *use case sequence diagram* in Fig. 15. As a simplification, we assume that the Cashier controls switching between normal and express mode, in the end it makes no difference who does it. In this sequence diagram, *max* denotes the maximum number of items that the current sale can process. It is a fixed number if the use case is in the express checkout mode and infinity otherwise.

The protocol that the sequence diagram defines is specified by the set of traces represented by the following regular expression:

$$
\begin{array}{l}
tr(SD_{uc1}) = \\
(\quad (enableExpress \quad (startSale\ enterItem^{(max)}\ finishSale\ cashPay)^*\ ) \\
\quad + \quad (disableExpress \quad (startSale\ enterItem^*\ finishSale\ (cashPay\ +\ cardPay))^*\ )\ )^*
\end{array}
$$

These traces are accepted by the state diagram given in Fig. 17 (note that the labels of states only serve as documentation. They are not UML compliant). We assume that the *ExMode* guard is initialized non-deterministically in the $Init$ state.

Functionality specification

We now start to analyze the functionality of each of the operations in the use case. An informal understanding of the functionality is to identify the classes, their properties, and to construct an initial class diagram, see Fig. 16. For the specification of the operations we assume:

1. There exists a *Store* object, *store : Store*.
2. The object *store* owns a set of *Product* objects with their barcode, amount, and price, denoted by *store.catalog*. It accesses the attribute *catalog : set(Product)* (we omit other properties not directly relevant to modelling, like product descriptions).
3. The *Cashdesk* object accesses the store via an association *store*.
4. There exists a *Clock* object associated with the *desk* via the association *clock*.
5. There is a *Bank* class with a method *authorize(Card c,double a;bool returns)*, which checks a credit card transaction with amount a and returns whether it is valid.

We now specify the functionality of the methods in *Cashdesk*.

Use Case **UC 1: Process Sale**

Class *Cashdesk*

Method *enableExpress()*
pre: *true*
post: *ExMode' = true*

Method *disableExpress()*
pre: *true*
post: *ExMode' = false*

Method *startSale()*
pre: *true*
post: /* a new *sale* is created, and its *line items* initialized to empty, and the date correctly recorded */
$sale' = Sale.New(false/complete,\ empty/lines,\ clock.date()/date)$

Method *enterItem(Barcode c, int q)*
pre: /* there exists a product with the input barcode *c* */
$store.catalog.\text{find(c)} \neq null$
post: /* a new *line* is created with its *barcode c* and *quantity q* */
$line' = LineItem.New(c/barcode, q/quantity)$
; $line.subtotal' = store.catalog.\text{find}(c).price \times q$
; $sale.lines.\text{add}(line)$

Method *finishSale()*
pre: *true*
post: $sale.complete' = true$
$\wedge\ sale.total' = sum[[l.subtotal \mid l \in sale.lines]]$

Method *cashPay(double a; double c)*
pre: $a \geq sale.total$
post: $sale.pay' = CashPayment.New(a/amount,\ a\text{-}sale.total/change)$
/* the completed *sale* is logged in *store*, and */
; $store.sales.\text{add}(sale)$; /* the inventory is updated */
$\forall l \in sale.lines,\ p \in store.catalog \bullet (\textbf{if}\ p.barcode = l.barcode\ \textbf{then}\ p.amount' = p.amount - l.quantity)$
; $store.sales.\text{add}(sale)$;
$\forall l \in sale.lines,\ p \in store.catalog \bullet (\textbf{if}\ p.barcode = l.barcode\ \textbf{then}\ p.amount' = p.amount - l.quantity)$

Invariants

A *class invariant* is established on initialization of an instance, i.e. through the constructor. It must hold after each subsequent method call to that class. We specify correct initialization of the cash desk as a *class invariant*:

Class Invariant *Cashdesk* : $store \neq null \wedge store.catalog \neq null$
$\wedge\ clock \neq null \wedge bank \neq null$

Static and dynamic consistency

When we have the constituents of the specification document: *class diagram*, *state diagram*, *sequence diagram* and the *functional specification*, we need to make sure that they are consistent [CLM07]. The *static consistency* of the requirement model is ensured by checking that

1. all types used in the specification are given in the class diagram,
2. all data attributes of any class used in the specification are correctly given in the class diagram,
3. all properties are correctly given as atrributes or associations in the class diagram, and the multiplicities are determined according to whether the type of the property is $set(C)$ or $bag(C)$ for some class C.
4. each method given in the functional specification is used in the other diagrams according to its signature, that is, the arguments (and return values) and their types match.
5. expressions occurring as guards are (type) consistent with their functional specifications, i.e., of right type, initialised before first use, etc.

Dynamic consistency

means that the dynamic flow of control and the interaction protocol are consistent:

1. If the actors follow the interaction protocol when interacting with the use case controller, the state diagram of the use case should ensure that the interaction is not blocked by guards. Formally speaking, the traces of method calls defined by the sequence diagram should be accepted by the state machine.
2. On the other hand, the traces that are accepted by the state diagram should be allowable interactions in the protocol defined by the sequence diagram.

The above two conditions are formalised and checked as trace equivalence between the sequence diagram and the state diagram in FDR [Sch00, Ros98]. We point the interested reader to [RW06] and [NB03] for more detailed applications of CSP to different flavours of state diagrams. However, we note that the reasons for having a sequence diagram and a state diagram are different:

- the denotational trace semantics for the sequence diagram is easy to use as the specification of the protocol in terms of temporal order of the events,
- the state diagram has an operational semantics which is easier to use for for verification of both safety and liveness properties.

The event-based sequence diagrams and state diagrams abstract the data functionality away and thus make checking practically feasible—i.e., naive model-checking of an OO program would require considering all possible values for attributes and arguments of methods.

Detailed Design

At this point, we illustrate the refinement rules of rCOS (see Subsec. 2) to the operations that were specified for the use cases in the previous section. We take each operation of each use case and decompose it, assigning functionalities to use cases according to attributes of classes. This happens mainly through application of the refinement rule for functional decomposition, called the *expert pattern*.

Navigation in the functional specification will be translated to setters and getters for attributes, and direct access for associations.

Occasionally, refinement will not directly introduce a concrete implementation, but may also lead to refinement on the functionality specification level. For an example, observe how the handling of sets in the following examples evolves first through further refinement before being eventually modeled in code.

Refinement of UC 1 & 2

We successively handle the previously specified operations. The refinement of the mode handling to code is trivial. We remind the reader that according to the problem description changing the physical light will be handled by a separate component.

`class` *Cashdesk*::	*enableExpress()*	{ *exmode := true* }
	disableExpress()	{ *exmode := false* }

The *startSale()* operation is refined by making the *Cashdesk* instance invoke the constructor of the *Sale* class. As the *Clock* is an entity located in the *Cashdesk*, we have to pass the current *Date* as an argument. This follows the *expert pattern*:

```
class Cashdesk::   startSale() { sale:=Sale.New(clock.date()) }
class Sale::       Sale(Date d)
                    { date := d; complete := false; total := 0; lines := empty }
```

In Java, sets are implemented as a class that *implements* the *interface Collection*. The constructor of the set class initializes the instance as an empty set. The formal treatment of set operations like *find()*, *add()*, and constructors in general is given in the existing rCOS literature. Thus, the constructor *Sale()* can be further refined to the following code:

```
class Sale:: Sale(Date d)
      { date := d; complete := false; total := 0; lines := set(LineItem).New() }
```

However, when design of a significant algorithm is required, such as calculating the greatest common divisor of two integer attributes or finding the shortest path in a directed graph object, the specification of the algorithm instead of code can be first designed in the refinement. For operation *enterItem()*, the precondition is checked by finding the product in the catalog that matches the input bar code. From the refinement rule for the *expert pattern*, the *navigation path store.catalog*.find() indicates the need for a method *find()* in the use case handler, that calls a method *find()* which in turn calls the method *find()* of the set *catalog*. Thus, we need to design the following methods in the relevant classes:

```
class Cashdesk::      find(Barcode code; Product returns) { store.find(code; returns) }
class Store::         find(Barcode code; Product returns) {catalog.find(code; returns)}
Class set(Product)::  Method find(Barcode code; Product returns)
                        Pre   ∃p : Product • (p.barcode = code ∧ contains(p))
                        Post  returns.barcode' = code
```

Applying the expert pattern to the navigation paths *line.subtotal*, *store.catalog*.find() and *sale.lines*.add(), we can refine the specification of *enterItem()* to:

```
class Cashdesk::   enterItem(Barcode code, int qty) {
                    if find(code) ≠ null then {
                      line:=LineItem.New(code, qty);
                      line.setSubtotal(find(code).price × qty);
                      sale.addLine(line)
                    } else { throw exception e() }   }
class Sale::       addLine(LineItem l) { lines.add(l) }
class LineItem::   setSubtotal(double a) { subtotal:=a }
```

Note that we use exception handling to signal that the precondition is violated. This allows us to introduce more graceful error handling later through refinement. This is different to translating the condition into an `assert` statement which would terminate the application, as this would preclude refinement.

We now refine method *finishSale()* using the expert pattern and define a method *setComplete()* and a method *setTotal()* in class *Sale*. These methods then will be called by the use case handler class.

`class` *Cashdesk*::	*finishSale()*	{ *sale.setComplete()*; *sale.setTotal()*; }
`class` *Sale*::	*setComplete()*	{ *complete*:=*true* }
	setTotal()	{ *total*:=*lines.sum()* }

For *cashPay()*, we need the total of the sale to check the precondition, accordingly we define *getTotal()* in class *Sale*. To create a payment, we define a method *makeCashPay()* called by the cash desk, and creates an object of type *CashPayment*. For logging the sale, we define a method *addSale()* in class *Store* that is called by the cash desk, that will use the method *add()* of the set of sales.

For updating the inventory, the universal quantification will be implemented by a loop, so we defer the implementation to a helper method:

`class` *Cashdesk*::	*cashPay(double amount; double return)* {
	if (*amount* $\geq$ *sale.getTotal()*) **then** {
	sale.makeCashPay(amount; return);
	store.addSale(sale);
	updateInventory() /* defined separately */
	} **else** { **throw** exception e(*amount* $\geq$ *sale.getTotal()*)}

`class` *Sale*::	*getTotal(; double returns)* { *returns* := *total* } *makeCashPay(double amount; double returns)* { *payment*:=*CashPay.New(amount)*; *returns*:=*getChange()* } *getChange(; double returns)* { *returns* := *amount* - *total* }
`class` *Store*::	*addSale(Sale s)* { *sales.add(s)* }

Recall the functional specification corresponding to *updateInventory()*:

Class *Cashdesk*::
$\forall l \in sale.lines,\ p \in store.catalog \bullet ($ **if** $p.barcode = l.barcode$ **then** $p.amount' = p.amount - l.quantity)$

It involves universal quantification over elements of a set. Such a specification is usually covered by some *design pattern*. The solutions always require loop statements, which, in an object-oriented setting, are for example covered by (Java) *iterators*, or they might be implemented in a database.

A design pattern is to first define a method for changing the variables, i.e., to update the amount of the product in the catalog. This implies a method *update(int qty)* in class *Product*, and then a method *update(Barcode code, int qty)*

in *catalog* whose type is *set(Product)* and which implements the loop for the quantification on p (we consider the iteration over *sale.lines* in the next step):

```
class Product::       update(int qty) { amount := amount-qty }
class set(Product)::  update(Barcode code, int qty) {
                        Iterator i := iterator();
                        while (i.hasNext()) {
                          Product p := i.next();
                          if p.barcode=code then p.update(qty);
                        }
class Store::         update(Barcode code, int qty) { catalog.update(code,qty) }
```

The quantification on *sale.lines* is then designed as another loop in the class of the method that contains the formula in its specification:

```
class Cashdesk::   updateInventory() {
                     Iterator j := sale.lines.iterator();
                     while (j.hasNext()) {
                       LineItem l := j.next();
                       store.update(l.barcode,quantity)
                   } }
```

Now we can also give an equivalent, more direct encoding of the two quantifications, where the inner loop is for the objects whose state is being modified by the specification.

```
class Cashdesk::   updateInventory() {
                     Iterator j := sale.lines.iterator();
                     while (j.hasNext()) {
                       LineItem l := (j.next();
                       /* inlined store.update()/catalog.update() call: */
                       Iterator i := store.catalog.iterator();
                       while  (i.hasNext()) {
                         Product p := i.next();
                         if p.barcode=l.barcode then p.update(l.quantity)
                       } } }
```

In *cardPay()*, the precondition invokes the function *authorize(Card, double)* of the *Bank*. We reuse *addSale(sale)* and *updateInventory()* unchanged from the refinement for *cashPay()*. At this stage, where the *Bank* is an external class we do not need to specify the *authorize(Card, double)* method.

```
class Cashdesk::   cardPay(Card c) {
                   if (Bank.authorize(c,sale.total)) then  {
                     payment:=CardPay.New(c);
                     store.addSale(sale);
                     updateInventory()
                   } else { throw exception e(c)};
```

The other use cases expand in a similar way. The refinement of specifications involving universal and existential quantifications over a collection of objects/-data to Java implementation of the *Collection interface* show that formal methods should now take the advantages of the libraries of the modern programming languages such as Java. This can significantly reduce the burden on (or the amount of) verification.

Component-Based Architecture

The component architecture is designed from the object-oriented models in the previous sections. In contrast to the component layout in Chapter 3, where already deployment has been taken into account for the component mapping, we will first map the object-oriented model to logical components, and then discuss how they are affected by deployment. Also, we have some *a priori* components, like the hardware devices and the *Bank*.

The adaptation of the object-oriented model to a component-based model is made to *reduce system coupling*, such that less related functionalities are performed by different components. This is done according to use cases and users (i.e. actors).

Logical model of the component-based architecture

The primary use case **UC 1** is performed by the the *SalesHandler* component, while the composition of the handler with the components for the peripherals yields the *CashDesk*. A *Store* component aggregates several *CashDesk*s and an *Inventory*.

For the other use cases, we obtain a similar structure with a controller and some supporting classes (not shown in detail): Ordering stock (**UC 3**), handling deliveries (**UC 4**), stock report (**UC 5**), and changing prices (**UC 7**) are components within a *Store*.

Delivery reports (**UC 6**) are generated inside the *Enterprise* component, while product exchange between stores (**UC 8**) is managed in the *Exchange* component, which resides within *Enterprise*.

The model is called a *model of the logical component-based architecture* because

1. it is the model of the design for the application components,
2. the interfaces are object-oriented interfaces, that is interactions are only through local object method invocations.

However, it is important to note that the object-oriented functional refinement are needed for the identification of the components and their interfaces.

We take some liberties with the design of Chapter 3: we do not model a *CashDeskLine*, but only a single cash desk that accesses the inventory. Also, we omit the *CashBox* as it does not contribute to the presentation. Note that in the following, only the *SalesHandler* is actually derived from the requirements (as would be the *Clock*, the *Bank*, and the *Inventory*).

The *SalesHandler* component will be the "work horse" of our cash desk. It implements the actual *Sale* use case protocol and also provides the necessary API for accessing the ongoing sale from the GUI. As a simplification, we assume that they can happen atomically at anytime, that is, the `pure` keyword indicates that the methods calls can be interleaved with those from the protocol. The provided protocol corresponds to the trace given in the Functional Description of the **UC 1**. The method invocations on the required side are derived (manually) from the refinement of the functional specification. The multiple `update` call-outs stem from the iteration when a sale is logged and the inventory updated.

component SalesHandler
required interface ClockIf { date() }
required interface BankIf { authorize(..) }
required interface StoreIf { update(..), find (..), addSale(..) }
provided pure interface SaleIf
provided interface CashdeskIf { getItem(..), getSubTotal(..), getTotal (..), getPayment() }
protocol { ([?enableExpress (?startSale date! (?enterItem find!)$^{(max)}$?finishSale
?cardPay authorize! addSale!)*
| ?disableExpress (?startSale date! (?enterItem find!)* ?finishSale
[?cardPay authorize! addSale! update!*
| ?cashPay addSale! update!*])*])* }
class *Cashdesk* **implements** SaleIf, CashdeskIf

Design of the concrete interaction mechanisms

After obtaining the model of the logical component-based architecture, we can replace the object-oriented interfaces by different interaction protocols according to the requirement descriptions and the physical locations of the components.

There are more than one *CashDesk* component instance, each having its own clock and sharing one *Inventory* instance per store. The interaction between them can then be implemented asynchronously using an event channel. RMI or CORBA can be used for interactions between a *Store* component and the *Enterprise* component.

If we decompose the *Inventory* into sub-components (the three layer architecture): the *application layer*, the *data representation layer* and the *database*, we can

- keep the OO interface between the application layer and data representation layer,

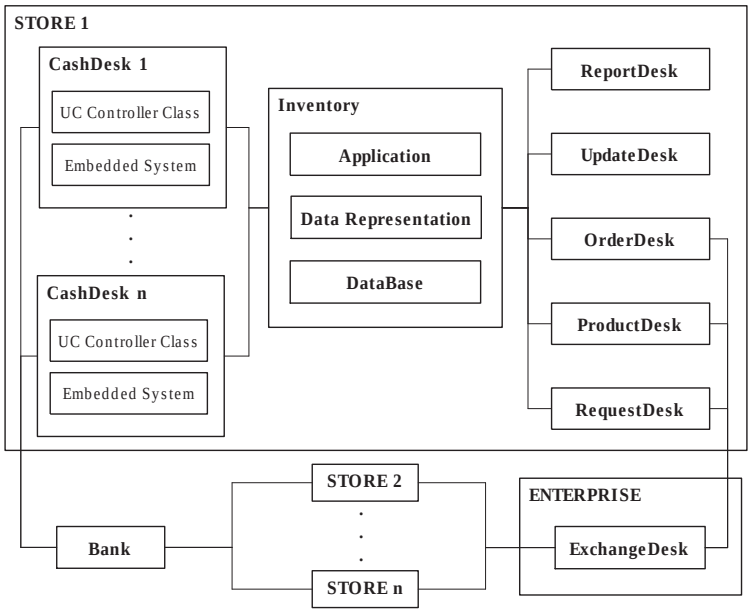


Figure 18: Overall component view of the system

- implement interaction between the data representation and the database in JDBC.

Most of these interactions mechanisms are international standards and the change from an OO interface to any of them has been a routine practice. We believe that there is no need for providing their formal models and analysis, though a formal method like rCOS is able to do this with some effort. Fig. 18 gives the overall view of the system. In the following, we discuss the detailed decomposition of the peripherals of a cash desk.

Hardware components

The peripheral device components are modelled in rCOS only at the *contract* level, that is, with regard to the protocols. We do not give their functional description or implementation here and assume they implement their behaviour correctly. The required protocols (call-outs with trailing exclamation mark; see [HLL05]) have been derived from the functional specifications/refinement.

The input devices are modelled as (active) rCOS processes that call provided methods of another component on input. For manual input, we model the cash desk terminal as a *black box* (we dispense with the implementing class) with buttons for starting/ending a sale, and manual input of an item and its quantity. Recall that we designed the controller class to handle both express mode changes. Nonetheless, here we stick to the original problem description and allow the cashier

to disable it only. Thus, the protocol is still a subset of the one induced by the use case (we omit method signatures for conciseness of the presentation):

```
// define short−hand for methods
define SaleIf { enableExpress(), disableExpress(), startSale (), enterItem (..),
                finishSale (), cardPay (..), cashPay(..) }

component Terminal
required interface SaleIf
protocol { ([disableExpress!] startSale ! enterItem!* finishSale ! [cardPay! | cashPay!])* }
```

Furthermore, we assume that the bar code scanner has the same interface (although it will in practice only ever invoke the *enterItem()* method). To connect both devices to the cash desk application, we have to introduce a *controller* which merges input from both devices. For later composition, we introduce unique names to the two provided interfaces of the same type and specify the class which handles the call-ins (implementation not shown). Here, we give the *combined* required/provided protocol of call-ins (methods prefixed by `?`)/call-outs (suffixed by `!`). rCOS also permits separate protocols for a component interface, which does not reveal any dependencies on method calls.

```
component InputController
required interface SaleIf
provided interface SaleIf at PortA, PortB // interleaving
protocol { ( [?disableExpress disableExpress!] ?startSale startSale ! // relay messages
             // fan in from both devices:
             (?enterItem enterItem!)*
             ?finishSale finishSale ! [ ?cardPay cardPay! | ?cashPay cashPay!])* }
class Merge implements SaleIf
```

The cash desk display provides a way of updating the display with the current sale. For each event, the display controller queries the cash desk's current sale via getter-methods and updates the screen. The interface will be provided by the *SaleHandler* component. Also, we handle displaying the mode here. Note that the GUI has a more general protocol as we do not need to take mode changes into account for an individual sale.

```
component CashDeskGUI
required interface LightIf { lightExpress(), lightNormal() }
required interface CashdeskIf { getItem(..), getSubTotal(..), getTotal (..), getPayment() }
required interface ClockIf { date }
provided interface GUIIf { enterItem(..), startSale(), finishSale (), cardPay (..),
                           cashPay (..), enableExpress(), disableExpress() }
protocol { ( [?enableExpress lightExpress!| ?disableExpress lightNormal!] ?startSale date!
             (?enterItem getItem! getSubTotal!)* ?endSale getTotal!
             [?cardPay | ?cashPay] getPayment! )* }
class GUI implements GUIIf
```

The *Printer* component shall employ the same design, providing *Printer* and *PrinterIf*.

As the system should use a bus architecture, updates have to be done in an event-based fashion, i.e., we need a *BusController* component that proxies between all devices and acts as a fan-out when an event has multiple subscribers. Contrary to the design document, we do not employ a broadcast architecture: for example,

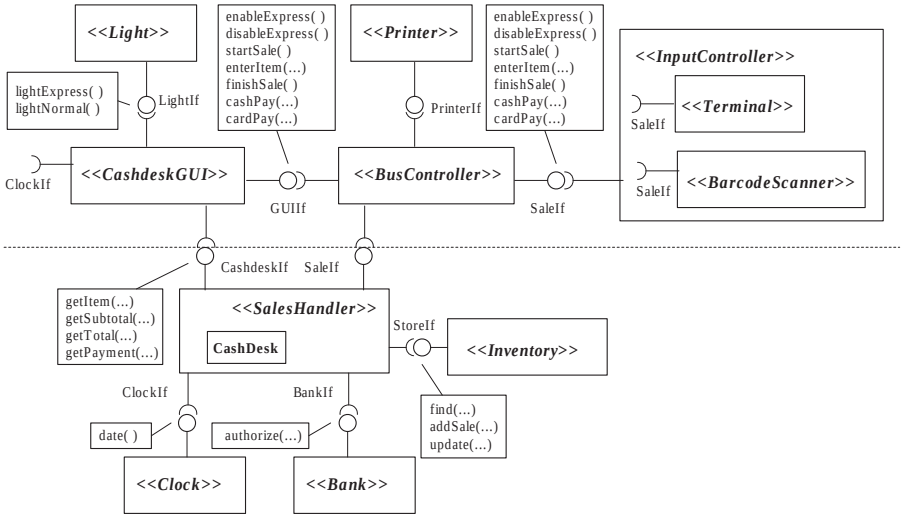


Figure 19: Deployed components for a single Cashdesk

the controller makes sure that the business logic processes an `enterItem` event *first*, and only *then* notifies the display. Likewise, it drives the printer.

component BusController
provided interface SaleIf *// to InputController*
required interface CashDeskGUI.GUIIf *// elided, see above*
required interface Printer.PrinterIf *// ditto*
required interface SaleIf *// from business logic*
protocol { /∗ fan−out for each call−in elided ∗/ }
class Bus **implements** SaleIf

We concede that this component design means that the *BusController* must be modified each time a new subscriber is added to the system.

We plumb the *BarcodeScanner* and *Terminal* component into the *InputController*, which we connect to the *BusController*. That is in turn connected to the *SalesHandler*. We omit detailed discussion of the other interfaces; dependent components mentioned in `with`-clauses are deployed automatically as long as there are no ambiguities with regard to interfaces:

component Cashdesk
  deploy CashDeskGUI **with** Clock, Light
  deploy InputController **with** Barcode **at** PortA, Terminal **at** PortB
  deploy BusController **with** InputController, CashDeskGUI, Printer
  deploy SalesHandler **with** Clock,Bank,Inventory,CashDeskGUI,BusController

Assuming availability of the required components of the *SalesHandler*, the resulting component is *closed*, as all required interfaces are provided. For the resulting component diagram, see Fig. 19; the upper half indicates the peripherals, the lower half the components derived from the use case.

With regard to formal rCOS (see e.g. [CHLZ06, HLL06b]), we note that only components whose traces start with a call-in are *rCOS-components*. Those that

have a call-out at the start of their trace, are actually *rCOS-processes*. For **UC 1**, only the input devices used by the actor are processes.

Modelling deployment

For a consistent rCOS model, this means we must modify the existing model to take into account the *deployment boundaries*, *middle-ware* and their effect on communication object references.

We also note that the different failure modes of remote communication must be taken into account and may require to revisit the Design, as for example, suddenly functions (in the mathematical sense) may *fail* when they are invoked on remote hosts. This is a field of ongoing investigation.

4 Analyses

This section outlines how to add the specification and analysis of the extra functionalities given in the description document. It then continues with a description of the actual analyses of the functional and behavioural properties that have been carried out with tool support.

Extra-functional Properties

We specify extra functionality of a method as a property for the time interval for the execution of the method. We use temporal variables whose value depend on the reference time interval for the execution of methods for our specification. Those variables could be ET_m which is the duration of the execution of method m in the worst case, or $N\_Customers$ which is the number of customers in the referenced observation time interval. From the intended meaning, the variable ET_m is rigid, its value does not depend on the reference interval. For a formula f on the rigid and temporal variables, for a probability p, $[f]_p$ is a formula saying that f is satisfied with the probability p. As it is well-known in the interval logic, the formulas $\phi;\psi$, which corresponds to the sequential composition of formulas ϕ and ψ, holds for an interval $[a,b]$ iff there is $m \in a..b$ such that ϕ holds for interval $[a,m]$ and ψ holds for interval $[m,b]$. Let ℓ be a temporal variable denoting the length of the interval it applied to. Intuitively, the formula

$[0 \leq ET\_ScanItem < 5]_{0.9} \wedge [0 \leq ET\_ScanItem < 1.0]_{0.05}$

says that the execution time for the operation *ScanItem* is within 5 seconds with probability 0.9, and it is less than 1 second with probability 0.05.

Since the arrival and leaving rates are the same: 320/3600 arrivals per second, and constant, with an exponential distribution,we can derive that $[N\_Customers = \frac{2}{45}\ell]_1$ holds for all intervals. As another example, by estimating the average waiting time for customers here we show how to include QoS analysis in our framework. Let $ET\_Service$ stand for the average service time for customers. It is

easy to calculate the possibility of *ET_Service* from the above specification, i.e. $ET\_Service = 32.075$. Therefore, the rate of service is $\mu = 1/ET\_Service = 1/32.075 = 0.0311$. Also, we have that the rate of customers arriving is $\lambda = N\_Customers/\ell = 4/45$.

Verification, Analysis and Tool support

Various verifications and analyses are carried out on different models. For the requirement model, the trace equivalence between the sequence diagram and its state diagram has been experimentally checked with FDR. We manually checked the consistency between the class declarations (i.e. the class diagrams) and the functionality specification to ensure that all classes and attributes are declared in the class declarations. This is obviously a syntactic and static semantic check that can be automated in a tool. We can further ensure the consistency by translating the rCOS functionality specification into a JML specification and then carry out runtime checking and testing. Also, some of the development steps involving recurrent patterns can be automated.

Runtime checking and testing in JML

We have not checked the correctness of the design against the requirement specification for removing possible mistakes made when manually applying the rules. However, we have translated some of the design into JML [Lea06] and carried out runtime testing of specifications and the validity of an implementation.

We translate each rCOS class C into two JML files, one is *C.jml* that contains the specification translated from the rCOS specification, and the other is a Java source file *C.java* containing a code that implements the specification. During the translation, the variables used in the rCOS specification are taken as specification-only variables in *C.jml*, that are mapped to program variables in *C.java*. The translated JML files can be compiled by the JML Runtime Assertion Checker Compiler (*jmlc*). Then, test cases can be executed to check the satisfaction of the specification by the implementation. The automatic unit testing tool of JML (*jmlunit*) can be used to generate unit testing code, and the testing process can be executed with *JUnit*.

For example, a JML code snippet of the *enterItem*() design in Section 3 is shown on the left of Fig. 20. Notice that the code in the dotted rectangle gives the specification of the exception that was left unspecified in Section 3.

The final code implementing the *enterItem*() specification is shown on the right of Fig. 20. Before getting the final code, we encountered two runtime errors reported by the testing process. One error resulted from the implementation which did not handle an input that falsifies the precondition. The reason for the other error is that one invariant is false after method execution. Testing is not sufficient for correctness. Therefore, it is also desirable to carry out static analysis, for instance with ESC/Java [CKLP06].

```
/*@ public normal_behaviour
  @   requires (\exists Object o; theStore.theProductList.contains(o);
  @                ((Product)o).theBarcode.equals(code)); …
  @   ensures   theLine != \old(theLine) &&
  @    theLine.theBarcode.equals(code) &&…
  @ also
  @ public exceptional_behaviour
  @     requires !(\exists Object o; theStore.theProductList.contains(o);
  @                ((Product)o).theBarcode.equals(code));
  @    signals_only Exception;
  @*/
public void enterItem(Barcode code, int quantity) throws Exception;
```

```
public void enterItem(Barcode code, int quantity)
        throws Exception{
   line = new LineItem(code, quantity);
   Iterator it = store.productList.iterator();
   boolean t = false;
   while (it.hasNext()){
      Product p = (Product)it.next();
      if (p.barcode.equals(code)){
         line.total = p.price * quantity;
         t = true;
         sale.lines.add(line);
      }
   }
   if (!t)   throw new Exception();
}
```

Figure 20: JML Specification and Implementation.

QVT transformation support

Our long term goal is to implement correctness preserving transformations that support a full model driven development process. The problems we are concerned with are the consistency among models on the same level, and the correctness relation between models in different levels. The meaning of consistency among models on the same level is that the models of various views need to be syntactically and semantically compatible with each other. The meaning of correctness relation between models on different levels is that a model must be semantically consistent with its refinements [LMRY06].

We plan to use QVT [Gro05], a model transformation language standard by OMG, to implement these model transformations. We have already defined the required rCOS metamodels for object diagrams, object sequence diagrams, component diagrams, component interaction diagrams and state machines. Pre- and post-conditions can also be translated into the respective clauses of a QVT program.

The refinement of the use cases on the object level through the expert pattern is done manually now, but it can be implemented using QVT, and automated. The correctness of the expert pattern is proved by rCOS. We have already explored correctness preserving transformations in a object-oriented design in [YMSL06].

Then we can apply architectural design to decompose the object model into a component model by allocating use cases, classes, associations and services to components. The component model should be a refinement of the application requirement model. This step can also be implemented as a QVT transformation. The correctness of the transformation from object model to component model should be proved in rCOS.

Verifying interaction protocols

Composition of components not only requires that the interfaces and their types match. Also, the interaction protocols must be compatible; if two interfaces are composed, the corresponding traces must match, i.e., the sequences of call-ins/call-outs must align: unexpected call-ins are ignored by the callee and will deadlock

the caller.

We automatically check protocol consistency by generating CSP processes for each interface. Interface composition is then modelled through pairwise parallel composition. As a composed interface is uniquely defined in the specification, we can successively check each composition for deadlock freedom and incrementally add successive interfaces. Model checking with FDR would indicate a deadlock if an operation call required by some component is not provided by any of its partner components at some moment in time.

In an application, these can also be implemented as runtime checks using extensions for aspect-oriented programming that capture temporal behaviour like Tracematches [AAS+05] or Tracechecks [BS06].

5 Summary

We have presented our modelling of the Common Component Example in rCOS, the Relational Calculus of Object and Component Systems. Based on the problem description, we have developed a set of interrelated models for each use case which separately models the different concerns of control and data. The rigorous approach ensures that we can be of *high confidence* that the resulting *program* implements the desired behaviour *correctly* without having to prove this on the generated code, which usually is very difficult or even impossible. As the problem description is not always amenable to modelling in rCOS, we occasionally had to simplify the model.

For each use case, a state diagram, a sequence diagram, its trace and the functional specification of its operations with pre- and postconditions are provided. These different aspects shall help all participants involved in the development process (designers, programmers) to share the same overall understanding of the system. Consistency of models is checked through processes that can be automated, e.g. by type checking of OO methods and model checking of traces.

The functional specifications in rCOS are then refined to a detailed design very close to Java code through *correct* rules for patterns like the Expert Pattern or translation of quantification. The generated code can be enriched with JML annotations derived from the functional specification and invariants. The annotations can then be used for runtime checking or static analysis.

From the OO model, we then derive a more convenient component model using Class Decomposition and grouping classes into components. The rCOS component model allows us to reason about component interaction, defined by the traces from the specifications, ruling out "bad" behaviour like deadlocks. We discuss issues of (distributed) deployment and necessary middle-ware.

For extra-functional analysis, we applied the Probabilistic Interval Temporal Logic to specify extra-functional properties given in the problem description. Then, we conducted the estimation of the average waiting time for customers.

Apart from concrete tool support, we also point out ongoing work and research on automating the different parts of the development process.

The generated code and additional information is available from the project web page at `http://www.iist.unu.edu/cocome/`.

Acknowledgements

The authors thank Wang Xu and Siraj Shaikh at UNU-IIST for helpful suggestion on CSP and FDR, and the careful reviewers.

1 Full CSP/FDR listing for consistency

```
-- $Id: cocome.fdr2,v 1.6.2.2 2007/06/25 05:58:09 vs Exp $
-- Define events:
channel enableExpress, disableExpress, startSale, enterItem, finishSale, cashPay, cardPay

-- Define the process corresponding to the regular expression:
Trace = (TraceExMode [] TraceNormalMode) ; Trace
TraceNormalMode = disableExpress -> TraceNormalSale
TraceNormalSale = startSale -> enterItem -> TraceEnterItemLoopStar
                ; finishSale -> ((TraceCashPay [] TraceCardPay)
                ; (SKIP [] TraceNormalSale))
TraceEnterItemLoopStar = SKIP [] (enterItem -> TraceEnterItemLoopStar)
TraceCashPay = cashPay -> SKIP
TraceCardPay = cardPay -> SKIP

TraceExMode = enableExpress -> TraceESale
TraceESale = startSale -> enterItem -> TraceEMode(7)
           ; (finishSale -> (TraceCashPay ; (SKIP [] TraceESale)))
TraceEMode(c) = if c == 0 then SKIP
                else (SKIP [] (enterItem -> TraceEMode(c-1)))

-- State Diagram:
datatype Mode = on | off
State = Init(on) [] Init(off)
-- Resolve outgoing branches non-deterministically:
Init(mode) = (if mode == on then disableExpress -> StateNormalMode(off)
                            else STOP)
           [] (if mode == off then enableExpress -> StateExpressMode(on)
                              else STOP)
StateNormalMode(mode) = (startSale -> enterItem -> StateEnterItemLoopStar)
                      ; finishSale -> ((StateCashPay [] StateCardPay)
                      ; ((enableExpress -> StateExpressMode(on))
                      [] StateNormalMode(mode)))

StateEnterItemLoopStar = SKIP [] (enterItem -> StateEnterItemLoopStar)
StateCashPay = cashPay -> SKIP
StateCardPay = cardPay -> SKIP

StateEMode(c) = if c == 0 then SKIP
                else (SKIP [] (enterItem -> StateEMode(c-1)))

StateExpressMode(mode) = startSale -> enterItem -> StateEMode(7)
                       ; finishSale -> (StateCashPay
                       ; ((disableExpress -> StateNormalMode(off))
                       [] StateExpressMode(mode)))
```

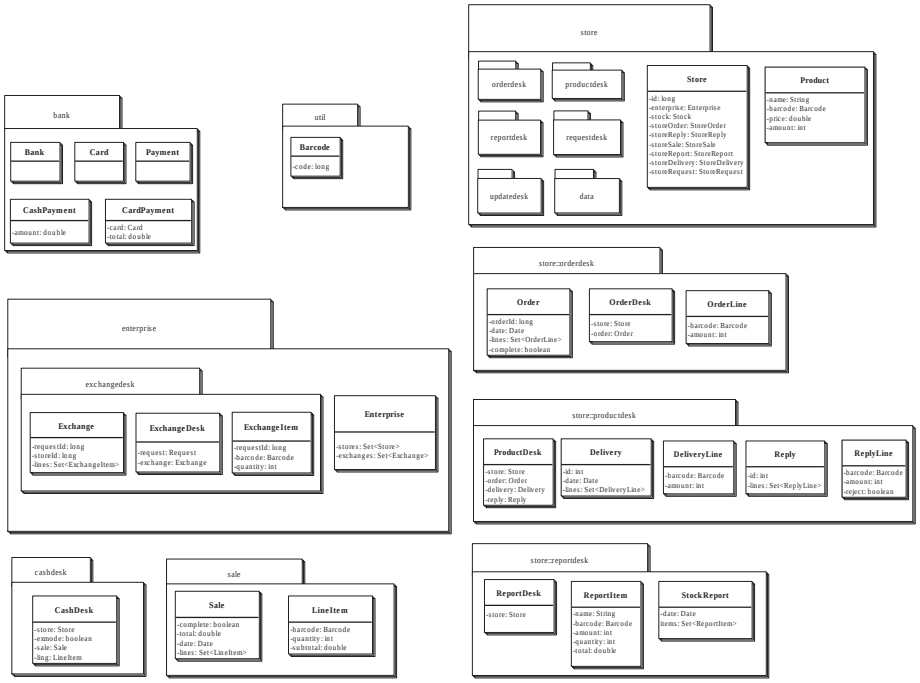


Figure 21: Packages

−− Check trace equivalence:
assert State [T= Trace
−−ˆ does not hold **as** trace abstracts from the guard,
−− permits: enableExpress −> ... −> enableExpress
assert Trace [T= State

−− Make sure both mechanisms can't deadlock:
assert Trace :[deadlock free [F]]
assert State :[deadlock free [F]]

2 Packages

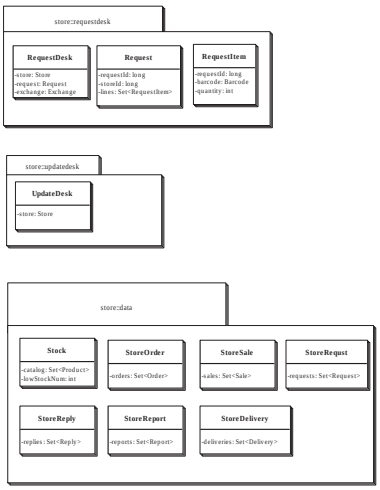


Figure 22: Packages

Paper F: Analyzing Orchestration of BPEL Specified Services with Model Checking[6]

Authors:

Joseph C. Okika

Aalborg University, Denmark

Abstract

This project investigates, implements and evaluates tool support for analysis of SOA-Based service contracts using Model Checking. The specification language for the contract is Business Process Execution Language (BPEL). It captures the behavior of services and allows developers to compose services without dependence on any particular implementation technology. A behavior specification is extracted from a BPEL program for formal analysis. One of the key conditions is that it reflects the intended semantics for BPEL, and in order to make it comprehensible, it is specified in a functional language. The resulting tool suite is hosted on an Eclipse platform.

Keywords: Analysis, Modeling checking, BPEL, Service contract, and SOA

[6]This chapter is previously published in [Oki09].

1 Research Question and Its Significance

Service Oriented Architectures (SOAs) are applicable when multiple applications running on varied technologies and platforms need to communicate with each other. In this way, enterprises can mix and match services to perform business transactions with less programming effort. However, a service operates under a contract/agreement which will set expectations, and a particular ontological standpoint that influences its semantics [PL03]. Services are first class citizens and are autonomous as well as distributed in nature. They can be composed to form higher level services or applications to solve business goals. Of course, this raises a lot of issues such as managing composed services, monitoring their interaction, analyzing the behavior of interacting services, verifying the functionality of individual services as well as composed services.

So far, service development has used traditional testing which are inefficient when dealing with distributed systems. Thus, there is a clear need to employ and integrate successful analysis techniques like model checking in the design of support tools for effectively solving these problems as well as in the implementation of high quality SOA-based services. Therefore, a detailed contractual description of services and corresponding semantics is of great importance.

BPEL offers a programming model for specifying the orchestration of web services through several activities. Activities are categorized into two; basic and structured. Basic activities (for instance invoke, receive, etc.) define the interaction capabilities of BPEL processes whereas the structured activities are made up of constructs such as flow (for synchronization), compensate, and pick among other activities.

Current Results In [DPC+05] we have demonstrated a viable solution to the problem of checking for some functional and behavioural properties of individual services. This is done through translation of the specifications to timed automata followed by model checking for relevant properties. In [COR07] we consider the problem of consistency across specifications and identified a need to set up a correspondence between the individual automata. The novel contribution in that paper is to make such a consistency check practical by translating the automata to CCS, the input language for the Concurrency Work Bench. As demonstrated by a case study, this technique is applicable and gives a handle for automating yet another consistency check for web services.

2 Current knowledge and the existing solutions

In this section, we present several efforts geared toward formalizing/analyzing services specified using BPEL - one of the most widely used orchestration language.

The overall observation about these works is that they all deal with three major issues; semantics definition, mapping to an analysis language and applicability. In Figure 23, δ represents those efforts that cover semantics definition and mostly applying Petri net simulation while μ represents those that focus on mapping/-translation to an analysis language.

The issue with most of these results is that they cover only fragments of the language and for some of them, there is not explicit statement about the underlying analysis language and possibility of automation.

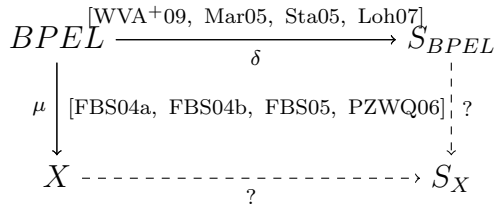


Figure 23: BPEL Formalizations

Abstract state machines are used in [FR05] to define an abstract operational semantics for BPEL for version 1.1. The work focuses on formal verification of service implementations and resolving inconsistencies in the standard. Abouzaid and Mullins [AM08] propose a BPEL-based semantics for a new specification language based on the π-calculus, which will serve as a reverse mapping to the π-calculus based semantics introduced by Lucchi and Mazzara [LM07b]. Their mapping is implemented in a tool integrating the toolkit HAL and generating BPEL code from a specification given in the BP-calculus. Unlike in our approach, this work covers the verification of BPEL specifications through the mappings while the consistency of the new language and the generated BPEL code is yet to be considered. As a future work, the authors plan to investigate a two way mapping.

Several model checking approaches have been employed to provide some form of analysis. An overview of most of the semantics foundation is given in [vBK06]. An illustrative example which is well-explained is [Mar05]. It deals with specification of both the abstract model and executable model of BPEL. The approach is based on Petri nets where a communication graph is generated representing a process's external visible behavior. It verifies the simulation between concrete and abstract behavior by comparing the corresponding communication graphs. Continuing with Petri net, an algebraic high-level Petri net semantics of BPEL is presented in [Sta05]. The idea here is to use the Petri patterns of BPEL activities in model checking certain properties of BPEL process descriptions. The model is feature complete for BPEL 1.1. Lohmann extends this work with a feature-complete Petri net semantics for BPEL 2.0 [Loh07].

As there exists several BPEL formalizations including a comprehensive and rigorously defined mapping of BPEL constructs onto Petri net structures presented in [WVA+09, OVvdA+07] a detailed comparison and evaluation of Petri

Net semantics for BPEL is presented in [LVOS08]. The comparison reveals different modelling decisions with a discussion on their consequences together with an overview of the different properties that can be verified on the resulting models.

In the case of using labeled transition systems as models for formalizing BPEL, few efforts is found in the literature which focuses on some fragments of BPEL constructs. For instance, Geguang et al. present a language μ-BPEL [PZWQ06] where a full operational semantics using a labeled transition system is defined for this language and its constructs to Extended Timed Automata. The language constructs are mapped to a simplified version of BPEL 1.1. Fu et al. presented a translation from BPEL to guarded automata in their work [FBS04a]; the guarded automata is further translated into Promela specification which is the language for the SPIN model. Similar approaches are also followed in [FBS04b, FBS05]. All these efforts points to the fact that there is an important need for service contracts to be specified and analyzed.

3 Proposed Ideas

As mentioned in the previous section, many theoretical results have considered the semantics analysis of BPEL. However, there are several issues around these semantics. First, there is the issue of coverage - that is to say, is the full BPEL language covered or some fragments of it? Most of the efforts using automata as presented in the previous section covers only some fragments of BPEL. A few of the efforts using Petri net covers a feature-complete BPEL. We point this out because it is worthwhile to have a comparison with another full coverage in a different formalism like automata. Second, there is the issue of translation where one may ask: is it semantic preserving? There is also the third issue of whether it is manual, semi-automated or automated.

Figure 24 shows the proposed approach; mapping BPEL to timed automata (TA). This defines a semantics for BPEL with a clear description of what is included and what is abstracted in the mapping and thus answers the issues raised above.

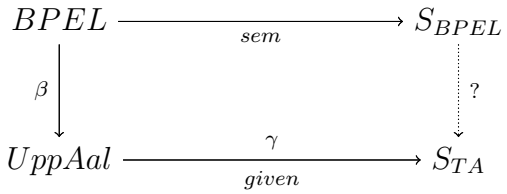


Figure 24: The new Approach

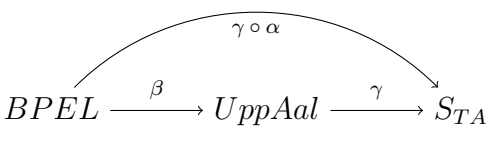


Figure 25: Functional Composition

Looking at Figure 24, starting from BPEL, we consider a full behavior of BPEL syntax and define the semantics based on UppAal, S_{BPEL}. We follow a

functional approach where we define a function β mapping BPEL to UppAal. We use timed automata for the formal model but with a rendering to UppAal because it is a mature model checking tool with wider audience and supported in our research environment. It can be a different choice (for example SPIN) in another environment. Note that the function *given* which takes care of the TA semantics is given with the UppAal tool and it's transition system semantics. Composing these two functions as shown in Figure 25 relates $BPEL$ to S_{TA}. In effect, having defined the function mapping BPEL to UppAal, we achieve a semantic preserving extraction/translation. That is, taking the inverse of the function gives us the result.

In [OR08] we give a classification of service contract specification languages based on application families and aspects. The classification identifies competing languages across aspects. It shows where a language may fit into the development of service based applications as well as the ones that allow for desired analyses, for instance match of functionality, protocol compatibility or performance match. In addition, we use the classification to survey analysis approaches. Furthermore, the classification may assist in planning of development activities, where an application involves services with contracts that span across families. Such scenarios are to be expected as service oriented applications spread. Another paper [COR08] focuses on analyzing behavioral properties for web service contracts formulated in Business Process Execution Language (BPEL) and Choreography Description Language (CDL). The key result reported is an automated technique to check consistency between protocol aspects of the contracts. The contracts are abstracted to (timed) automata and from there a simulation is set up, which is checked using automated tools for analyzing networks of finite state processes.

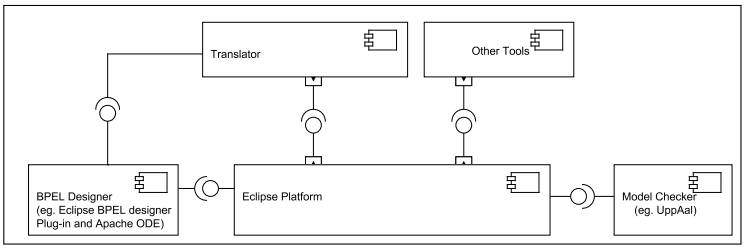


Figure 26: Plug-in Architecture

4 Contribution to the problem domain and Discussion

The project offers three distinct contributions in the development of analysis and verification tools for SOA-based services. 1) The technique employed is a rigorous

use of the power of functional languages in defining a property preserving mapping for the full behavior of BPEL. 2) Model checking of behavior properties of BPEL. General properties such as those related to deadlock and reachability as well as application specific properties are considered. Eg., services should not deadlock even with faults and compensation. 3) A prototype Integrated tool. The supporting tool will allow developers to leverage the already existing IDE such as Eclipse to design, specify and analyze SOA-based services.

Tool Development: We focus on building a theory based tool that gives developers of SOA-based services a clear understanding of BPEL processes. We are implementing the integrated supporting tool as a plug-in in the Eclipse framework. A model (UML) of the various components of the analysis tool is shown in Figure 26.

Discussion: The main novelty is to solve the issue of semantics unrelated to analysis tools. This is achieved by defining the extraction function using a functional language. As a side effect to this, we develop a functional XML parser/unparser for Standard ML. As this is an ongoing work, further effort will be geared toward tuning the tool. We plan to build a service based point of sale system using ActiveVOS orchestration system to demonstrate analysis of properties.

Paper G: Operational Semantics for BPEL Complex Features in Rewriting Logic[7]

Authors:
Joseph C. Okika
Aalborg University, Denmark

Olaf Owe and Cristian Prisacariu
University of Oslo, Norway

[7]This chapter is previously published in [OOP09].

1 Motivation

Business Processing Execution Language (BPEL) is a core part of the Web Services protocol and a service orchestration language for Service Oriented Architecture (SOA). Formal semantics of full BPEL is under intense research in the SOA community because of the inherent difficulties of some complex (and important) features of BPEL. These features include the nested concurrency model and the combination of communication, exception handling, and shared resources. Apart from the common concurrency problems, such as deadlock or race condition, one needs to consider that BPEL supports developing applications with multiple services running in parallel and equipped with a transaction mechanism called compensation. Further, a service in this setting does not have a full knowledge of services that it will interact with in advance. These lead to synchronization problems, e.g.: how do different services synchronize on data? How are faults (exceptions) in nested concurrency handled?

The efforts invested into finding a formal semantics for BPEL have several motivations: a formalization of BPEL would provide better understanding of the language, and would facilitate building of tools for simulation, analyses, and verification of systems with mathematical precision. Among the formalizations found in the literature, those based on Petri nets offer support for simulation [Loh07, OVvdA+07], whereas those based on labeled transitions systems offer support for automatic verification [FBS05, PZWQ06, MR08]. However, because the complex features of BPEL are different from the conventional programming languages, these formalizations do not fully capture the meaning of these features.

In our work we take an operational semantics approach and formalize some of the complex features of BPEL based on rewriting logic [Mes92]. The advantage of this approach is that the semantics is now executable in Maude [CDE+02]. This caters for analyses of BPEL processes using simulation and for verification using breath-first search.

2 Complex Features of BPEL

A BPEL process models the public behavior of interacting partner services or the private behavior of a partner service in a business interaction. It specifies when to wait for messages, and when to send messages among other activities. BPEL processes are programmed using the following constructs:

Declarations of process variables, partner links, correlation sets, etc. are done at a process level, in addition to handlers (without compensation and termination). For a scope, all scope declarations apply at the scope level.

Scope declares variables similar to a process including compensation and termination handlers. It provides a context for processing these declarations and activities.

concurrency	::=	$flow\ \ \mathcal{A}_1\ \ \mathcal{A}_2\ \ ...\ \ \mathcal{A}_n$
compensation	::=	$compensate\ \ s \mid compensateScope\ \ s$
scope	::=	$scope\ \ decl\ \ s\ \ \mathcal{A}\ \ eventhandler^*\ \ faulthandler^*$ $compensationhandler\ \ terminationhandler$
bproc	::=	$decl\ \ eventhandler^*\ \ faulthandler^*\ \ \mathcal{A}$
eventhandler	::=	$p\ \ \mathcal{A} \mid d\ \ \mathcal{A}$
$faulthandler$	::=	$F'\ \ \mathcal{A}$ where $F' \subseteq F$
compensationhandler	::=	$\mathcal{A}$
terminationHandler	::=	$\mathcal{A}$

Table 5: Abstract Syntax for concurrent flow, process, scope and handlers

Handlers (`<eventHandlers>`, `<faultHandlers>`, `<compensationHandler>`, `terminationHandler`) can be defined for the process or for a scope to manage exceptions.

Compensation is used to manage long-running transactions by reversing successfully completed activities. Compensations can be activated by `<compensate>` and `<compensateScope>` activities.

Activity includes the basic activities (`<assign>`, `<validate>`, `<empty>`, `<throw>`, `<rethrow>`, `<wait>`) for data manipulation and service interaction; the structuring activities (`<sequence>`, `<while>`, `<if>`, `<repeatUntil>`, `<forEach>`) for organizing activities; and the concurrent activity (`<flow>`) for capturing concurrency.

Among all these, the more complex and important ones that we deal with are listed in Table 5 (in abstract syntax). The $flow$ construct captures the concurrency aspect of BPEL. The other constructs are for *scope* and various handlers for managing exceptions. An event handler can be a message based event (receiving a message on a specified port p) or an alarm based event (timeout of a specified time d). In the case of the fault handlers there can be a specific fault (handled by a *catch* clause in BPEL) or a set of faults (handled by a *catchall* clause) hence the $F' \subseteq F$ in the syntax definition, where F is the set of fault names, $\mathcal{A}$ denotes the set of activities, and s a scope name.

3 Operational Semantics Approach using Rewriting Logic

Rewriting logic and its tool Maude provide a semantical framework and a high-level language for defining operational semantics for languages; and have been

used to formalize a number of modeling languages and paradigms. The facilities for prototyping and tool support are valuable when experimenting and comparing different semantic definitions. An advantage of rewriting logic is that it has a general, built-in notion of concurrency and readily supports different mechanisms for interaction, including object-oriented systems and message passing systems. Concurrency is typically based on defining multi-sets of units, which become concurrent when the left hand sides of the rewriting logic rules concern only one unit. Conceptually, several units are rewritten at the same time by applying the rules to all of them independently.

In order to formalize BPEL one needs the notion of nested concurrency, embedded in nested scopes of handlers, and variable declarations. Furthermore, interaction is based on both events, exceptions and shared resources. These features call for a a very general framework. In our work we will show that the Maude concurrency model can be adapted to the setting of nested systems in a natural way. A BPEL configuration consists of an environment reflecting the scope structure, including the binding of entities and the context of handlers, together with a program continuation and inner configurations representing concurrent processes at that scoping level. Event-based communication can then be formalized with true concurrency semantics, in the sense of non-overlapping rule applications, whereas shared resources give rise to non-deterministic interleaving of the actions depending on a shared resource. In this way the main challenges of BPEL are solved in a unified manner. BPEL processes can then be simulated by running the operational semantics in Maude with the BPEL processes as input. The set of all possible executions can be explored by the search command; both breath-first search and bounded breath-first search.

Bibliography

[AAA+07] Alexandre Alves, Assaf Arkin, Sid Askary, Ben Bloch, Francisco Curbera, Yaron Goland, Neelakantan Kartha, Sterling, Dieter König, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, and Alex Yiu. Web Services Business Process Execution Language Version 2.0. OASIS Standard, 2007.

[AAS+05] C. Allan, P. Avgustinov, A.S. Simon, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittamplan, and J. Tibble. Adding Trace Matching with Free Variables to AspectJ. In *OOPSLA '05*, pages 345–364, 2005.

[AAS06] M.A. Aslam, S. Auer, and J. Shen. From BPEL4WS Process Model to Full OWL-S Ontology. Poster, Third European Semantic Web Conference, 2006.

[ACD+03] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. *BPEL4WS, Business Process Execution Language for Web Services Version 1.1.* IBM, 2003.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[ADLH+04] B. Atkinson, G. Della-Libera, S. Hada, M. Hondo, P. Hallam-Baker, J. Klein, B. LaMacchia, P. Leach, J. Manferdelli, H. Maruyama, et al. Web Services Security (WS-Security), Version 1, OASIS Standard 200401. Technical report, OASIS, 2004.

[AM08] Faisal Abouzaid and John Mullins. A Calculus for Generation, Verification and Refinement of BPEL Specifications. *ENTCS*, 200(3):43–65, 2008.

[BBC+06] S. Bajaj, D. Box, D. Chappell, F. Curbera, G. Daniels, P. Hallam-Baker, M. Hondo, C. Kaler, D. Langworthy, A. Malhotra, et al. Web Services Policy Framework (WS-Policy). Technical report, IBM, BEA Systems, Microsoft, SAP AG, Sonic Software, VeriSign, 2006.

[BGP06] L. Baresi, S. Guinea, and P. Plebani. WS-Policy for service monitoring. In *6th VLDB Intl. Workshop on Technologies for E-Services*, volume 3811 of *LNCS*, pages 72–83. Springer, 2006.

[BHES07] B. Bordbar, G. Howells, M. Evans, and A. Staikopoulos. Model Transformation from OWL-S to BPEL Via SiTra. In David H. Akehurst, Régis Vogel, and Richard F. Paige, editors, *Proceedings ECMDA-FA*, volume 4530 of *LNCS*, pages 43–58. Springer, 2007.

[BHL+04] Mark Burstein, Jerry Hobbs, Ora Lassila, Drew Mcdermott, Sheila Mcilraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. OWL-S: Semantic Markup for Web Services. Website http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/, 2004.

[BJP99] Antoine Beugnard, Jean-Marc Jézéquel, and Noël Plouzeau. Making components contract aware. *IEEE CS Press*, 32(7):38–45, 1999.

[BL06] David Booth and Canyang K. Liu. Web Services Description Language (WSDL) Version 2.0 Part 0: Primer. Candidate recommendation, W3C, 2006.

[BLL+96] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal in 1995. *Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 431–434, 1996.

[BS06] Eric Bodden and Volker Stolz. Tracechecks: Defining semantic interfaces with temporal logic. In Welf Löwe and Mario Südholt, editors, *5th Intl. Symp. on Software Composition (SC'06)*, volume 4089 of *LNCS*, pages 147–162. Springer, 2006.

[CCK+01] J. Clark, C. Casanave, K. Kanaskie, B. Harvey, J. Clark, N. Smith, J. Yunker, and K. Riemer. ebXML Business Process Specification Schema Version 1.01. Technical report, OASIS, 2001.

[CCLP06] S. Carpineti, G. Castagna, C. Laneve, and L. Padovani. A formal account of contracts for Web Services. In *WS-FM, 3rd Int.l Workshop on Web Services and Formal Methods*, number 4184 in LNCS, pages 148–162. Springer, 2006.

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. *Web Services Description Language (WSDL) 1.1.* W3C, 1.1 edition, March 2001. URL: http://www.w3c.org/TR/wsdl.

[CDE+02] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[CDE+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. Introduction. In *All About Maude*, pages 1–28. Springer, 2007.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. The MIT Press, 1999.

[CGP07] G. Castagna, N. Gesbert, and L. Padovani. A Theory of Contracts for Web Services. In *PLAN-X '07, 5th ACM-SIGPLAN Workshop on Programming Language Technologies for XML*, pages 37–48, 2007.

[CHH+08] Zhenbang Chen, Abdel Hakim Hannousse, Dang Van Hung, Istvan Knoll, Xiaoshan Li, Yang Liu, Zhiming Liu, Qu Nan, Joseph C. Okika, Anders P. Ravn, Volker Stolz, Lu Yang, and Naijun Zhan. Modelling with relational calculus of object and component systems–rCOS. In A. Rausch, R. Reussner, R. Mirandola, and Frantisek Plasil, editors, *The Common Component Modeling Example*, volume 5153 of *LNCS*, chapter 3, pages 116–145. Springer, 2008.

[CHLZ06] X. Chen, J. He, Z. Liu, and N. Zhan. A Model of Component-Based Programming. Technical Report 350, UNU-IIST, P.O. Box 3058, Macao SAR, China, 2006. http://www.iist.unu.edu, Accepted by FSEN'07.

[CKLP06] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures*, volume 4111 of *LNCS*, pages 342–363. Springer, 2006.

[Cla98] James Clark. XSL Transformations (XSLT) Version 1.0. Technical Report REC-xml-19980210, W3C, 1998. `http://www.w3.org/TR/xslt`.

[CLM07] X. Chen, Z. Liu, and V. Mencl. Separation of Concerns and Consistent Integration in Requirements Modelling. In *Proc. Current Trends in Theory and Practice of Computer Science*, volume 4362 of *LNCS*, pages 819–831. Springer, 2007.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[CNYM00] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, Boston/Dordrecht/London, 2000.

[COR07] Emilia Cambronero, Joseph C. Okika, and Anders P. Ravn. Analyzing Web Service Contracts - An Aspect Oriented Approach. In *Proceedings of the International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM'2007)*, pages 149 – 154. IEEE CS Press, November 2007.

[COR08] Emilia Cambronero, Joseph C. Okika, and Anders P. Ravn. Consistency Checking of Web Service Contracts. *International Journal On Advances in Systems and Measurements*, 1(1):29–39, 2008.

[CRNMB99] Sérgio Campos, Berthier Ribeiro-Neto, Autran Macedo, and Luciano Bertini. Formal verification and analysis of multimedia systems. In *MULTIMEDIA '99: Proceedings of the seventh ACM International conference on Multimedia (Part 1)*, pages 419–430, New York, NY, USA, 1999. ACM Press.

[DCP+06] G. Diaz, M. E. Cambronero, J. J. Pardo, V. Valero, and F. Cuartero. Automatic Generation of Correct Web Services Choreographies and Orchestrations with Model Checking Techniques. In *IEEE International Conference on Internet and Web Applications and Services ICIW'06*, volume 1257, pages 33–40. IEEE CS Press, 2006.

[DDK+04] A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kbler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef. Web Services on demand: WSLA-driven automated management. *IBM Systems Journal*, 43(1), 2004.

[DKR04] H. Davulcu, M. Kifer, and I. V. Ramakrishnan. CTR-S: A Logic for Specifying Contracts in Semantic Web Services. In *Proceedings of WWW2004*, pages 144–153, May 2004.

[DLDG+02] G. Della-Libera, B. Dixon, P. Garg, P. Hallam-Baker, M. Hondo, C. Kaler, H. Maruyama, A. Nadalin, N. Nagaratnam, A. Nash, et al. Web Services Trust Language (WS-Trust). Technical report, IBM, Verisign, Microsoft and RSA Security, 2002.

[DPC+05] G. Diaz, J. J. Pardo, M. E. Cambronero, V. Valero, and F. Cuartero. Verification of Web Services with Timed Automata. In *Proceedings of First International Workshop on Automated Specification and Verification of Web Sites*, volume 157, pages 19–34. Electronics Notes in Theoretical Computer Science - ENTCS, 2005.

[EK02] Wolfgang Emmerich and Nima Kaveh. Component technologies: Java beans, COM, CORBA, RMI, EJB and the CORBA component model. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 691–692, New York, NY, USA, 2002. ACM Press.

[EM95] Ásgeir Th. Eiríksson and Kenneth L. McMillan. Using formal verification/analysis methods on the critical path in system design: A case study. In *Proceedings of the 7th International Conference on Computer Aided Verification*, pages 367–380, London, UK, 1995. Springer.

[Erl05] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, 2005.

[Erl07] Thomas Erl. *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007.

[ES08] Christian Eisentraut and David Spieler. Fault, Compensation and Termination in WS-BPEL 2.0 - A Comparative Analysis. In *Proceedings of 5th International Workshop on Web Services and Formal Methods*, pages 107–126. Springer, 2008.

[FBS04a] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of interacting BPEL Web Services. In *Proc. 13th Int'l Conf. WWW'04*, pages 621–630, NY, USA, 2004. ACM Press.

[FBS04b] Xiang Fu, Tevfik Bultan, and Jianwen Su. WSAT: A Tool for Formal Analysis of Web Services. In *Proc. 16th Int. Conf. Comp. Aided Verification*, pages 510–514. Springer, 2004.

[FBS05] Xiang Fu, Tevfik Bultan, and Jianwen Su. Synchronizability of conversations among web services. *IEEE Transactions on Software Engineering*, 31(12):1042–1055, December 2005.

[FL77] Michael J. Fischer and Richard E. Ladner. Propositional modal logic of programs. In *STOC'77*, pages 286–294. ACM Press, 1977.

[FPO+09] Stephen Fenech, Gordon J. Pace, Joseph C. Okika, Anders P. Ravn, and Gerardo Schneider. On the specification of full contracts. *Electronic Notes Theoretical Computer Science - ENTCS*, 253(1):39–55, 2009.

[FPS08] Stephen Fenech, Gordon J. Pace, and Gerardo Schneider. Conflict analysis of deontic contracts (extended abstract). In *NWPT'08*, pages 34–36, November 2008.

[FR05] Dirk Fahland and Wolfgang Reisig. ASM-based semantics for BPEL: The negative Control Flow. In *Proc. 12th International Workshop on Abstract State Machines*, pages 131–151, 2005.

[GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In Umeshwar Dayal and Irving L. Traiger, editors, *SIGMOD Conference*, pages 249–259. ACM Press, 1987.

[GORS06] Pablo Giambiagi, Olaf Owe, Anders P. Ravn, and Gerardo Schneider. Language-Based Support for Service Oriented Architectures: Future Directions. In Joaquim Filipe, Boris Shishkov, and Markus Helfert, editors, *Proceedings of ICSOFT (1)*, pages 339–344, Setúbal, Portugal, September 2006. INSTICC Press.

[Gov05] Guido Governatori. Representing business contracts in RuleML. *International Journal of Cooperative Information Systems*, 14(2-3):181–216, 2005.

[GQ07] Susanne Graf and Sophie Quinton. Contracts for BIP: Hierarchical Interaction Models for Compositional Verification. In *FORTE*, pages 1–18, 2007.

[Gro05] Object Management Group. MOF QVT final adopted specification, ptc/05-11-01. http://www.omg.org/docs/ptc/05-11-01.pdf, 2005.

[HH98] C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[HL05] Reiko Heckel and Marc Lohmann. Towards Contract-based Testing of Web Services. *Electronic Notes in Theoretical Computer Science - ENTCS*, 116:145–156, 2005.

[HLL05] J. He, X. Li, and Z. Liu. Component-Based Software Engineering. In *Proc. ICTAC'2005*, volume 3722 of *LNCS*, pages 70–95. Springer, 2005.

[HLL06a] J. He, X. Li, and Z. Liu. rCOS: A refinement calculus for object systems. *Theoretical Computer Science*, 365(1-2):109–142, 2006.

[HLL06b] Jifeng He, Xiaoshan Li, and Zhiming Liu. A Theory of Reactive Components. In Z. Liu and L. Barbosa, editors, *Intl. Workshop on Formal Aspects of Component Software (FACS 2005)*, volume 160, pages 173–195. Electronics Notes in Theoretical Computer Science - ENTCS, 2006.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hol97] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[JP04] B. Jacobs and E. Poll. Java Program Verification at Nijmegen: Developments and Perspective. In *Software Security - Theories And Systems: Part 2: Verification of Security Propertiesl Symposium, Isss 2003, Tokyo, Japan*, volume 3233 of *LNCS*, pages 134–153. Springer, 2004.

[Jur06] M.B. Juric. *Business Process Execution Language for Web Services*. Packt Publishing, 2006.

[KBR+04] Nickolas Kavantzas, David Burdett, Gregory Ritzinger, Tony Fletcher, and Yves Lafon. Web Services Choreography Description Language Version 1.0. W3C working draft, W3C, December 2004. http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/.

[KGH+07] Anish Karmarkar, Martin Gudgin, Marc Hadley, Yves Lafon, Jean-Jacques Moreau, Henrik Frystyk Nielsen, and Noah Mendelsohn. SOAP version 1.2 part 1: Messaging framework (second edition). W3C recommendation, W3C, April 2007. http://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

[KL03] A. Keller and H. Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *Journal of Network and Systems Management*, 11(1):57–81, March 2003.

[KNT06] Tomohiro Kaizu, Tomoya Noro, and Takehiro Tokuda. A state propagation method for consistency checking of web service function invocations in web applications. In *ICWE '06: Workshop proceedings of the sixth international conference on Web engineering*, page 18, New York, NY, USA, 2006. ACM Press.

[KPS08] Marcel Kyas, Cristian Prisacariu, and Gerardo Schneider. Run-time monitoring of electronic contracts. In *ATVA'08*, LNCS. Springer, October 2008.

[KS76] J.G. Kemeny and J.L. Snell. *Finite Markov Chains*. Springer, 1976.

[KvB04] Mariya Koshkina and Franck van Breugel. Modelling and Verifying Web Service Orchestration by means of the Concurrency Workbench. *SIGSOFT Softw. Eng. Notes*, 29(5):1–10, 2004.

[Lam02] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[Law09] Loraine Lawson. What a BPEL Engine Brings to CERN's SOA Initiative, 2009. http://www.itbusinessedge.com/cm/community/features/interviews/blog/what-a-bpel-engine-brings-to-cerns-soa-initiative/?cs=23241.

[LBR06] G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[LE06] Z. Liu and J. He (Eds.). *Mathematical Frameworks for Component software: Models for Analysis and Synthesis, Series on Component-Based Software Development - Vol. 2.* World Scientific, 2006.

[Lea06] J.L. Leavens. JML's Rich, Inherited Specification for Behavioural Subtypes. In *Proc. 8th International Conference on Formal Engineering Methods (ICFEM06)*, volume 4260 of *LNCS*, pages 2–34. Springer, 2006.

[LGB05] S. Lippe, U. Greiner, and A. Barros. A Survey on State of the Art to Facilitate Modelling of Cross-Organisational Business Processes. In *In: Proceedings of the 2nd GI-Workshop XML4BPM 2005*, pages 7–21, 2005.

[Lin05] Peter F. Linington. Automating support for e-business contracts. *Int. J. Cooperative Inf. Syst.*, 14(2-3):77–98, 2005.

[LLZ06] X. Liu, Z. Liu, and L. Zhao. Object-oriented structure refinement - a graph transformational approach. Technical Report 340, UNU-IIST, P.O. Box 3058, Macao SAR, China, 2006. http://www.iist.unu.edu, Published in Proc. Intl. Workshop on Refinement, ENTCS.

[LM07a] Yves Lafon and Nilo Mitra. SOAP Version 1.2 Part 0: Primer (Second Edition). W3C recommendation, W3C, April 2007. http://www.w3.org/TR/2007/REC-soap12-part0-20070427/.

[LM07b] Roberto Lucchi and Manuel Mazzara. A pi-calculus based semantics for WS-BPEL. *Journal Log. Algebr. Program.*, 70(1):96–118, 2007.

[LMRY06] Z. Liu, V. Mencl, Anders P. Ravn, and L. Yang. Harnessing theories for tool support. Proceedings of International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA06), Full version as UNU-IIST Technical Report 343, http://www.iist.unu.edu, 2006.

[Loh07] Niels Lohmann. A feature-complete Petri net semantics for WS-BPEL 2.0. In Kees van Hee, Wolfgang Reisig, and Karsten Wolf,

editors, *Proc. Workshop on Formal Approaches to Business Processes and Web Services*, pages 21–35. University of Podlasie, June 2007.

[LPT08] Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. A Formal Account of WS-BPEL. In *Proceedings of 10th International Conference on Coordination Models and Languages*, LNCS, pages 199–215. Springer, 2008.

[Ltd] Formal Systems (Europe) Ltd. http://www.fsel.com/software.html.

[LVOS08] Niels Lohmann, H.M.W. Verbeek, Chun Ouyang, and Christian Stahl. Comparing and evaluating Petri net semantics for BPEL. *IJBPIM*, 2008.

[LX05] Xingwu Liu and Zhiwei Xu. Interaction Complexity - A Computational Complexity Measure for Service-Oriented Computing. In *SKG '05: Proceedings of the First International Conference on Semantics, Knowledge and Grid*, page 33. IEEE CS Press, 2005.

[Mar05] Axel Martens. Consistency between executable and abstract processes. In *EEE '05: Proceedings of the 2005 IEEE International Conference on e- Technology, e-Commerce and e-Service (EEE'05) on e-Technology, e-Commerce and e-Service*, pages 60–67, Washington, DC, USA, 2005. IEEE CS Press.

[Mes92] José Meseguer. Conditioned Rewriting Logic as a United Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mes98] J. Meseguer. Membership algebra as a logical framework for equational specification. In *In 12th International Workshop on Recent Trends in Algebraic Development Techniques (WADT'97)*, volume 1376 of *LNCS*, pages 18–61. Springer, 1998.

[Mes00] José Meseguer. Rewriting Logic and Maude: a Wide-Spectrum Semantic Framework for Object-Based Distributed Systems. In *Proceedings of Fourth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 89–. Kluwer, 2000.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 1997.

[MH05] Jan Mendling and Michael Hafner. From Inter-organizational Workflows to Process Execution: Generating BPEL from WS-CDL. In *OTM Workshops*, volume 3762 of *LNCS*, pages 506–515. Springer, 2005.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[MLM+06] C. Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter F. Brown, and Rebekah Metz. Reference Model for Service Oriented Architecture 1.0. WWW page, 2006.

[MOM02] Narciso Martí-Oliet and José Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.

[MORW04] M. Möller, E-R. Olderog, H. Rasch, and H. Wehrheim. Linking CSP-OZ with UML and Java: A case study. In *Proc. Integrated Formal Methods (IFM'04)*, volume 2999 of *LNCS*, pages 267–286. Springer, 2004.

[MR04] José Meseguer and Grigore Rosu. Rewriting logic semantics: From language specifications to formal analysis tools. In David A. Basin and Michaël Rusinowitch, editors, *IJCAR*, volume 3097 of *LNCS*, pages 1–44. Springer, 2004.

[MR07] José Meseguer and Grigore Rosu. The Rewriting Logic Semantics Project. *Theoretical Computer Science*, 373(3):213–237, 2007.

[MR08] Radu Mateescu and Sylvain Rampacek. Formal Modeling and Discrete-Time Analysis of BPEL Web Services. In *Proceeedings of 4th International Workshop on CIAO! and EOMAS*, Lecture Notes in Business Information Processing, pages 179–193, 2008.

[NB03] M.Y. Ng and M. Butler. Towards formalizing UML state diagrams in CSP. In *1st Intl. Conf. on Software Engineering and Formal Methods (SEFM'03)*, pages 138–147. IEEE CS Press, 2003.

[Oki09] Joseph C. Okika. Analyzing Orchestration of BPEL Specified Services with Model Checking. In *Proceedings of the PhD Symposium of the 7th International Joint Conference on Service Oriented Computing (ICSOC/ServiceWave)*, pages 1–6. CEUR-WS, 2009.

[OOP09] Joseph C. Okika, Olaf Owe, and Cristian Prisacariu. Operational Semantics for BPEL Complex Features in Rewriting Logic. In *Proceedings of the 21st Nordic Workshop on Programming Theory, NWPT '09*, pages 95–97, Kgs. Lyngby, Denmark, 2009. Danmarks Tekniske Universitet.

[OR08] Joseph C. Okika and Anders P. Ravn. Classification of SOA Contract Specification Languages. In *Proceedings of The IEEE International Conference on Web Services (ICWS)*, pages 433–440, Los Alamitos, CA, USA, September 2008. IEEE CS Press.

[OVvdA+07] Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H. M. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Science of Computer Programming*, 67(2-3):162–198, 2007.

[Pap03] M. P. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. In *4th International Conference on Web Information Systems Engineering (WISE)*, pages 3–12. IEEE CS Press, 2003.

[PH06] Michael P. Papazoglou and Willem-Jan Van Den Heuvel. Service-oriented design and development methodology. *Int. Journal Web Engineering Technology*, 2(4):412–442, 2006.

[PL03] Randall Perrey and Mark Lycett. Service-Oriented Architecture. In *Proceedings of 2003 Symposium on Applications and the Internet Workshops (SAINT)*, pages 116–119. IEEE CS Press, 2003.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science (FOCS'77)*, pages 46–57. IEEE CS Press, 1977.

[PPS07] Gordon Pace, Cristian Prisacariu, and Gerardo Schneider. Model Checking Contracts –a case study. In *Proceedings of 5th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 4762 of *LNCS*, pages 82–97. Springer, October 2007.

[PS07] Cristian Prisacariu and Gerardo Schneider. A formal language for electronic contracts. In Marcello Bonsangue and Einar Broch Johnsen, editors, *9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'07)*, volume 4468 of *LNCS*, pages 174–189. Springer, June 2007.

[PV02] F. Plasil and S. Visnosky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11):1056–1070, 2002.

[PvdH07] Michael P. Papazoglou and Willem-Jan van den Heuvel. Business process development life cycle methodology. *Communications of the ACM*, 50(10):79–85, 2007.

[PZ08] Christopher J. Pavlovski and Joe Zou. Non-Functional Requirements in Business Process Modeling. In Annika Hinze and Markus Kirchberg, editors, *Fifth Asia-Pacific Conference on Conceptual Modelling (APCCM 2008)*, volume 79 of *CRPIT*, pages 103–112, Wollongong, NSW, Australia, 2008. ACS.

[PZWQ06] Geguang Pu, Xiangpeng Zhao, Shuling Wang, and Zongyan Qiu. Towards the Semantics and Verification of BPEL4WS. *Electronic Notes in Theoretical Computer Science - ENTCS*, 151(2):33–52, 2006.

[RAA+05] Omer Rana, Asif Akram, Rashid Al Ali, David Walker, Gregor von Laszewski, and Kaizar Amin. *Extending Web Services Technologies: The Use of Multi-Agent Approaches (Multiagent Systems, Artificial Societies, and Simulated Organizations)*, chapter 8, pages 161–186. Springer, 2005.

[RDS07] Ervin Ramollari, Dimitris Dranidis, and Anthony J. H. Simons. A Survey of Service Oriented Development Methodologies. In *The 2nd European Young Researchers Workshop on Service Oriented Computing*, pages 75–80, 2007.

[RGWC99] D. Reeves, B. Grosof, M. Wellman, and H. Chan. Toward a Declarative Language for Negotiating Executable Contracts. In *AAAI-99 Workshop on Artificial Intelligence in Electronic Commerce (AIEC-99)*, 1999.

[Rom05] D. Roman. Web Service Modeling Ontology. *Applied Ontology*, 1(1):77–106, 2005.

[Ros98] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[RRMP08] Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil, editors. *The Common Component Modeling Example: Comparing Software Component Models [result from the Dagstuhl research seminar for CoCoME, August 1-3, 2007]*, volume 5153 of *LNCS*. Springer, 2008.

[RW06] A.W. Roscoe and Z. Wu. Verifying Statemate statecharts using CSP and FDR. In *8th Intl. Conf. on Formal Engineering Methods (ICFEM'06)*, volume 4260 of *LNCS*, pages 324–341. Springer, 2006.

[Sch00] Steve Schneider. *Concurrent and Real-time Systems*. Wiley, 2000.

[See01] Scott Seely. *SOAP: Cross Platform Web Service Development Using XML*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[SRM09] Traian-Florin Serbanuta, Grigore Rosu, and José Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*, 207(2):305–340, 2009.

[Sta05] Christian Stahl. A Petri Net Semantics for BPEL. Informatik-Berichte 188, Humboldt-Universitt zu Berlin, July 2005.

[Ste] Stephen Fenech. Full $\mathcal{CL}$ specification of CoCoME. Available from `http://www.cs.um.edu.mt/svrg/Tools/CLTool/Papers/CoCoMEfullCLSpec.pdf`.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, 2002.

[Taf07] Darryl K. Taft. Enterprise SOA Adoption to Double in Next Two Years. http://www.eweek.com/c/a/Application-Development/Enterprise-SOA-Adoption-to-Double-in-Next-Two-Years/, 2007. Last visited: March 11, 2008.

[tBBG06] M.H. ter Beek, A. Bucchiarone, and S. Gnesi. A Survey on Service Composition Approaches: From Industrial Standards to Formal Methods. Technical Report 2006-TR-15, Istituto di Scienza e Tecnologie dell Informazione A. Faedo (ISTICNR), Italy, 2006.

[Tha01] S. Thatte. XLANG: Web Services for Business Process Design. Technical report, Microsoft Corporation, 2001.

[Tos05] V. Tosic. On Comprehensive Contractual Descriptions of Web Services. In *IEEE International Conference on e-Technology, e-Commerce, and e-Service*, pages 444–449. IEEE CS Press, 2005.

[TPP02] Vladimir Tosic, Kruti Patel, and Bernard Pagurek. WSOL - Web Service Offerings Language. In Christoph Bussler, Richard Hull, Sheila A. McIlraith, Maria E. Orlowska, Barbara Pernici, and Jian Yang, editors, *Web Services, E-Business, and the Semantic Web, CAiSE 2002 International Workshop, WES 2002, Toronto, Canada, May 27-28, 2002, Revised Papers*, volume 2512 of *LNCS*, pages 57–67. Springer, 2002.

[vBK06] Frank van Breugel and Maria Koshika. Models and Verification of BPEL, 2006. http://www.cse.yorku.ca/ franck/research/drafts/tutorial.pdf.

[Vin97] S. Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *Communications Magazine, IEEE*, 35(2):46–55, 1997.

[Vir04] Mirko Viroli. Towards a Formal Foundation to Orchestration Languages. *Electronic Notes in Theoretical Computer Science - ENTCS*, 105:51–71, 2004.

[Viv09] Vivansa. XSIZE - Business Solution for EMCS, December 2009. http://www.vivansa.com/public/index.php?id=26.

[VRO+03] Joannis Vlachakis, Sascha Rex, Boris Otto, Markus Lebender, and Tom Fleckstein. Web Services: A look into quality and security aspects. Technical report, Fraunhofer-Institut für Arbeitswirtschaft und Organisation IAO, 2003.

[W3C04] Web Services Architecture. Technical report, World Wide Web Consortium, February 2004.

[WD06] D. Waldt and R. Drummond. ebXML: The Global Standard for Electronic Business. *Retrieved from http://www. ebxml. org*, 2006.

[Whi04] S.A. White. Introduction to BPMN. Technical report, 2004.

[WS-04] Web Services Agreement Specification (WS-Agreement). https://forge.gridforum.org/projects/graap-wg/document/WS-AgreementSpecification/en/7, 2004.

[wsa04] Web Services Architecture. W3C Working Group Note, `www.w3.org/TR/ws-arch/`, Feb 2004.

[WVA+09] M. T. Wynn, H. M. W. Verbeek, Aalst, Ter A. H. M. Hofstede, and D. Edmond. Business process verification - finally a reality! *Business Process Mgmt. Journal*, 15(1):74–92, 2009.

[YMSL06] L. Yang, V. Mencl, V. Stolz, and Z. Liu. Automating correctness preserving model-to-model transformation in MDA. In *Proc. of Asian Working Conference on Verified Software, UNU-IIST*, 2006. http://www.iist.unu.edu.

[ZK07] Yongyan Zheng and P. Krause. Automata Semantics and Analysis of BPEL. In *Digital EcoSystems and Technologies Conference, 2007. DEST '07. Inaugural IEEE-IES*, pages 147–152. IEEE CS Press, Feb. 2007.

[ZSGX05] Qiu Zongyan, Wang Shuling, Pu Geguang, and Zhao Xiangpeng. Semantics of BPEL4WS-like Fault and Compensation Handling. In *Proceedings of International Symposium of Formal Methods Europe*, volume 3582 of *LNCS*, pages 350–365. Springer, 2005.